

The L^AT_EX3 Sources

The L^AT_EX3 Project*

Released 2018-04-30

Abstract

This is the reference documentation for the `expl3` programming environment. The `expl3` modules set up an experimental naming scheme for L^AT_EX commands, which allow the L^AT_EX programmer to systematically name functions and variables, and specify the argument types of functions.

The T_EX and ϵ -T_EX primitives are all given a new name according to these conventions. However, in the main direct use of the primitives is not required or encouraged: the `expl3` modules define an independent low-level L^AT_EX3 programming language.

At present, the `expl3` modules are designed to be loaded on top of L^AT_EX 2 ϵ . In time, a L^AT_EX3 format will be produced based on this code. This allows the code to be used in L^AT_EX 2 ϵ packages *now* while a stand-alone L^AT_EX3 is developed.

While `expl3` is still experimental, the bundle is now regarded as broadly stable. The syntax conventions and functions provided are now ready for wider use. There may still be changes to some functions, but these will be minor when compared to the scope of `expl3`.

New modules will be added to the distributed version of `expl3` as they reach maturity.

*E-mail: latex-team@latex-project.org

Contents

I	Introduction to <code>expl3</code> and this document	1
1	Naming functions and variables	1
1.1	Terminological inexactitude	3
2	Documentation conventions	3
3	Formal language conventions which apply generally	5
4	<code>TeX</code> concepts not supported by <code>LaTeX3</code>	5
II	The <code>l3bootstrap</code> package: Bootstrap code	6
1	Using the <code>LaTeX3</code> modules	6
III	The <code>l3names</code> package: Namespace for primitives	7
1	Setting up the <code>LaTeX3</code> programming language	7
IV	The <code>l3basics</code> package: Basic definitions	8
1	No operation functions	8
2	Grouping material	8
3	Control sequences and functions	9
3.1	Defining functions	9
3.2	Defining new functions using parameter text	10
3.3	Defining new functions using the signature	11
3.4	Copying control sequences	14
3.5	Deleting control sequences	14
3.6	Showing control sequences	14
3.7	Converting to and from control sequences	15
4	Analysing control sequence names	16
5	Using or removing tokens and arguments	16
5.1	Selecting tokens from delimited arguments	18
6	Predicates and conditionals	19
6.1	Tests on control sequences	20
6.2	Primitive conditionals	20
V	The <code>l3expan</code> package: Argument expansion	22

1	Defining new variants	22
2	Methods for defining variants	23
3	Introducing the variants	24
4	Manipulating the first argument	26
5	Manipulating two arguments	27
6	Manipulating three arguments	28
7	Unbraced expansion	29
8	Preventing expansion	30
9	Controlled expansion	31
10	Internal functions	33
VI	The <code>l3tl</code> package: Token lists	34
1	Creating and initialising token list variables	34
2	Adding data to token list variables	35
3	Modifying token list variables	36
4	Reassigning token list category codes	36
5	Token list conditionals	37
6	Mapping to token lists	39
7	Using token lists	41
8	Working with the content of token lists	42
9	The first token from a token list	43
10	Using a single item	46
11	Viewing token lists	46
12	Constant token lists	46
13	Scratch token lists	47
VII	The <code>l3str</code> package: Strings	48
1	Building strings	48

2	Adding data to string variables	49
3	Modifying string variables	50
4	String conditionals	50
5	Mapping to strings	52
6	Working with the content of strings	54
7	String manipulation	57
8	Viewing strings	58
9	Constant token lists	59
10	Scratch strings	59
 VIII The l3quark package: Quarks		60
1	Quarks	60
2	Defining quarks	60
3	Quark tests	61
4	Recursion	61
5	An example of recursion with quarks	62
6	Scan marks	63
 IX The l3seq package: Sequences and stacks		65
1	Creating and initialising sequences	65
2	Appending data to sequences	66
3	Recovering items from sequences	66
4	Recovering values from sequences with branching	67
5	Modifying sequences	69
6	Sequence conditionals	69
7	Mapping to sequences	70
8	Using the content of sequences directly	71
9	Sequences as stacks	72

10	Sequences as sets	73
11	Constant and scratch sequences	74
12	Viewing sequences	75
X	The <code>l3int</code> package: Integers	76
1	Integer expressions	76
2	Creating and initialising integers	77
3	Setting and incrementing integers	78
4	Using integers	79
5	Integer expression conditionals	79
6	Integer expression loops	81
7	Integer step functions	83
8	Formatting integers	84
9	Converting from other formats to integers	85
10	Random integers	86
11	Viewing integers	86
12	Constant integers	87
13	Scratch integers	87
	13.1 Direct number expansion	88
14	Primitive conditionals	88
XI	The <code>l3flag</code> package: Expandable flags	90
1	Setting up flags	90
2	Expandable flag commands	91
XII	The <code>l3prg</code> package: Control structures	92
1	Defining a set of conditional functions	92
2	The boolean data type	94
3	Boolean expressions	96

4	Logical loops	98
5	Producing multiple copies	99
6	Detecting TeX's mode	99
7	Primitive conditionals	99
8	Nestable recursions and mappings	100
	8.1 Simple mappings	100
9	Internal programming functions	101
XIII The l3sys package: System/runtime functions		102
1	The name of the job	102
2	Date and time	102
3	Engine	102
4	Output format	103
XIV The l3clist package: Comma separated lists		104
1	Creating and initialising comma lists	104
2	Adding data to comma lists	106
3	Modifying comma lists	106
4	Comma list conditionals	107
5	Mapping to comma lists	108
6	Using the content of comma lists directly	110
7	Comma lists as stacks	110
8	Using a single item	112
9	Viewing comma lists	112
10	Constant and scratch comma lists	112
XV The l3token package: Token manipulation		113
1	Creating character tokens	113
2	Manipulating and interrogating character tokens	115

3	Generic tokens	118
4	Converting tokens	118
5	Token conditionals	119
6	Peeking ahead at the next token	122
7	Decomposing a macro definition	124
8	Description of all possible tokens	125
 XVI The <code>l3prop</code> package: Property lists		128
1	Creating and initialising property lists	128
2	Adding entries to property lists	129
3	Recovering values from property lists	129
4	Modifying property lists	130
5	Property list conditionals	130
6	Recovering values from property lists with branching	131
7	Mapping to property lists	131
8	Viewing property lists	132
9	Scratch property lists	133
10	Constants	133
 XVII The <code>l3msg</code> package: Messages		134
1	Creating new messages	134
2	Contextual information for messages	135
3	Issuing messages	136
4	Redirecting messages	138
5	Low-level message functions	139
 XVIII The <code>l3file</code> package: File and I/O operations		141
1	Input–output stream management	141
1.1	Reading from files	143

2	Writing to files	145
2.1	Wrapping lines in output	147
2.2	Constant input-output streams, and variables	148
2.3	Primitive conditionals	148
3	File operation functions	148
XIX	The l3skip package: Dimensions and skips	150
1	Creating and initialising dim variables	150
2	Setting dim variables	151
3	Utilities for dimension calculations	151
4	Dimension expression conditionals	152
5	Dimension expression loops	154
6	Dimension step functions	155
7	Using dim expressions and variables	156
8	Viewing dim variables	157
9	Constant dimensions	158
10	Scratch dimensions	158
11	Creating and initialising skip variables	158
12	Setting skip variables	159
13	Skip expression conditionals	160
14	Using skip expressions and variables	160
15	Viewing skip variables	160
16	Constant skips	161
17	Scratch skips	161
18	Inserting skips into the output	161
19	Creating and initialising muskip variables	162
20	Setting muskip variables	162
21	Using muskip expressions and variables	163
22	Viewing muskip variables	163

23	Constant muskips	164
24	Scratch muskips	164
25	Primitive conditional	164
XX	The l3keys package: Key–value interfaces	165
1	Creating keys	166
2	Sub-dividing keys	170
3	Choice and multiple choice keys	170
4	Setting keys	173
5	Handling of unknown keys	173
6	Selective key setting	174
7	Utility functions for keys	175
8	Low-level interface for parsing key–val lists	176
XXI	The l3fp package: Floating points	178
1	Creating and initialising floating point variables	179
2	Setting floating point variables	180
3	Using floating points	180
4	Floating point conditionals	182
5	Floating point expression loops	183
6	Some useful constants, and scratch variables	185
7	Floating point exceptions	186
8	Viewing floating points	187
9	Floating point expressions	187
9.1	Input of floating point numbers	187
9.2	Precedence of operators	189
9.3	Operations	189
10	Disclaimer and roadmap	196
XXII	The l3intarray package: fast global integer arrays	199

1	l3intarray documentation	199
1.1	Implementation notes	199
XXIII	The l3sort package: Sorting functions	200
1	Controlling sorting	200
XXIV	The l3tl-analysis package: Analysing token lists	201
1	l3tl-analysis documentation	201
XXV	The l3regex package: Regular expressions in T_EX	202
1	Syntax of regular expressions	202
2	Syntax of the replacement text	207
3	Pre-compiling regular expressions	209
4	Matching	209
5	Submatch extraction	210
6	Replacement	211
7	Constants and variables	211
8	Bugs, misfeatures, future work, and other possibilities	212
XXVI	The l3box package: Boxes	215
1	Creating and initialising boxes	215
2	Using boxes	216
3	Measuring and setting box dimensions	217
4	Box conditionals	217
5	The last box inserted	218
6	Constant boxes	218
7	Scratch boxes	218
8	Viewing box contents	218
9	Boxes and color	219

10	Horizontal mode boxes	219
11	Vertical mode boxes	220
12	Affine transformations	222
13	Primitive box conditionals	224
XXVII	The l3coffins package: Coffin code layer	226
1	Creating and initialising coffins	226
2	Setting coffin content and poles	226
3	Joining and using coffins	227
4	Measuring coffins	228
5	Coffin diagnostics	228
6	Constants and variables	229
XXVIII	The l3color-base package: Color support	230
1	Color in boxes	230
XXIX	The l3luatex package: LuaTeX-specific functions	231
1	Breaking out to Lua	231
2	Lua interfaces	232
XXX	The l3unicode package: Unicode support functions	233
XXXI	The l3candidates package: Experimental additions to l3kernel	234
1	Important notice	234
2	Additions to l3basics	235
3	Additions to l3box	236
3.1	Viewing part of a box	236
4	Additions to l3clist	236
5	Additions to l3coffins	237

6	Additions to <code>l3expan</code>	237
7	Additions to <code>l3file</code>	237
8	Additions to <code>l3flag</code>	238
9	Additions to <code>l3msg</code>	238
10	Additions to <code>l3prg</code>	240
11	Additions to <code>l3prop</code>	240
12	Additions to <code>l3seq</code>	241
13	Additions to <code>l3skip</code>	242
14	Additions to <code>l3sys</code>	243
15	Additions to <code>l3tl</code>	244
16	Additions to <code>l3token</code>	250
XXXII	The <code>l3drivers</code> package: Drivers	252
1	Box clipping	252
2	Box rotation and scaling	253
3	Color support	253
4	Drawing	253
4.1	Path construction	254
4.2	Stroking and filling	255
4.3	Stroke options	256
4.4	Color	256
4.5	Inserting \TeX material	257
4.6	Coordinate system transformations	257
XXXIII	Implementation	257
1	<code>l3bootstrap</code> implementation	257
1.1	Format-specific code	257
1.2	The <code>\pdfstrcmp</code> primitive in \XeTeX	258
1.3	Loading support Lua code	258
1.4	Engine requirements	259
1.5	Extending allocators	260
1.6	Character data	261
1.7	The \LaTeX 3 code environment	263
2	<code>l3names</code> implementation	264

3	Internal kernel functions	288
4	l3basics implementation	293
4.1	Renaming some TeX primitives (again)	293
4.2	Defining some constants	295
4.3	Defining functions	295
4.4	Selecting tokens	296
4.5	Gobbling tokens from input	298
4.6	Debugging and patching later definitions	298
4.7	Conditional processing and definitions	306
4.8	Dissecting a control sequence	311
4.9	Exist or free	313
4.10	Preliminaries for new functions	315
4.11	Defining new functions	317
4.12	Copying definitions	318
4.13	Undefining functions	319
4.14	Generating parameter text from argument count	319
4.15	Defining functions from a given number of arguments	320
4.16	Using the signature to define functions	321
4.17	Checking control sequence equality	323
4.18	Diagnostic functions	323
4.19	Doing nothing functions	325
4.20	Breaking out of mapping functions	325
5	l3expansion implementation	326
5.1	General expansion	326
5.2	Hand-tuned definitions	329
5.3	Definitions with the automated technique	332
5.4	Last-unbraced versions	333
5.5	Preventing expansion	334
5.6	Controlled expansion	335
5.7	Defining function variants	336
6	l3tl implementation	344
6.1	Functions	345
6.2	Constant token lists	347
6.3	Adding to token list variables	347
6.4	Reassigning token list category codes	349
6.5	Modifying token list variables	352
6.6	Token list conditionals	356
6.7	Mapping to token lists	360
6.8	Using token lists	362
6.9	Working with the contents of token lists	362
6.10	Token by token changes	364
6.11	The first token from a token list	366
6.12	Using a single item	371
6.13	Viewing token lists	371
6.14	Scratch token lists	372

7	l3str implementation	373
7.1	Creating and setting string variables	373
7.2	Modifying string variables	374
7.3	String comparisons	375
7.4	Mapping to strings	378
7.5	Accessing specific characters in a string	380
7.6	Counting characters	385
7.7	The first character in a string	386
7.8	String manipulation	387
7.9	Viewing strings	389
8	l3quark implementation	389
8.1	Quarks	389
8.2	Scan marks	392
9	l3seq implementation	393
9.1	Allocation and initialisation	394
9.2	Appending data to either end	397
9.3	Modifying sequences	398
9.4	Sequence conditionals	400
9.5	Recovering data from sequences	401
9.6	Mapping to sequences	404
9.7	Using sequences	407
9.8	Sequence stacks	407
9.9	Viewing sequences	408
9.10	Scratch sequences	409
10	l3int implementation	409
10.1	Integer expressions	410
10.2	Creating and initialising integers	412
10.3	Setting and incrementing integers	414
10.4	Using integers	415
10.5	Integer expression conditionals	415
10.6	Integer expression loops	419
10.7	Integer step functions	420
10.8	Formatting integers	422
10.9	Converting from other formats to integers	428
10.10	Viewing integer	431
10.11	Random integers	431
10.12	Constant integers	431
10.13	Scratch integers	433
10.14	Deprecated	433
11	l3flag implementation	433
11.1	Non-expandable flag commands	433
11.2	Expandable flag commands	434

12	l3prg implementation	436
12.1	Primitive conditionals	436
12.2	Defining a set of conditional functions	436
12.3	The boolean data type	436
12.4	Boolean expressions	438
12.5	Logical loops	442
12.6	Producing multiple copies	443
12.7	Detecting T _E X's mode	445
12.8	Internal programming functions	445
12.9	Deprecated functions	446
13	l3sys implementation	447
13.1	The name of the job	447
13.2	Time and date	447
13.3	Detecting the engine	447
13.4	Detecting the output	448
14	l3clist implementation	449
14.1	Removing spaces around items	449
14.2	Allocation and initialisation	451
14.3	Adding data to comma lists	453
14.4	Comma lists as stacks	453
14.5	Modifying comma lists	455
14.6	Comma list conditionals	458
14.7	Mapping to comma lists	459
14.8	Using comma lists	462
14.9	Using a single item	463
14.10	Viewing comma lists	465
14.11	Scratch comma lists	465
15	l3token implementation	466
15.1	Manipulating and interrogating character tokens	466
15.2	Creating character tokens	468
15.3	Generic tokens	472
15.4	Token conditionals	473
15.5	Peeking ahead at the next token	480
15.6	Decomposing a macro definition	485
15.7	Deprecated functions	486
16	l3prop implementation	486
16.1	Allocation and initialisation	488
16.2	Accessing data in property lists	490
16.3	Property list conditionals	494
16.4	Recovering values from property lists with branching	495
16.5	Mapping to property lists	495
16.6	Viewing property lists	496

17	l3msg implementation	497
17.1	Creating messages	497
17.2	Messages: support functions and text	498
17.3	Showing messages: low level mechanism	499
17.4	Displaying messages	502
17.5	Kernel-specific functions	510
17.6	Expandable errors	517
18	l3file implementation	519
18.1	Input operations	519
18.1.1	Variables and constants	519
18.1.2	Stream management	520
18.1.3	Reading input	522
18.2	Output operations	524
18.2.1	Variables and constants	524
18.3	Stream management	525
18.3.1	Deferred writing	527
18.3.2	Immediate writing	527
18.3.3	Special characters for writing	528
18.3.4	Hard-wrapping lines to a character count	528
18.4	File operations	537
18.5	Messages	543
18.6	Deprecated functions	544
19	l3skip implementation	546
19.1	Length primitives renamed	546
19.2	Creating and initialising <code>dim</code> variables	546
19.3	Setting <code>dim</code> variables	547
19.4	Utilities for dimension calculations	548
19.5	Dimension expression conditionals	549
19.6	Dimension expression loops	551
19.7	Dimension step functions	553
19.8	Using <code>dim</code> expressions and variables	554
19.9	Viewing <code>dim</code> variables	556
19.10	Constant dimensions	556
19.11	Scratch dimensions	556
19.12	Creating and initialising <code>skip</code> variables	557
19.13	Setting <code>skip</code> variables	558
19.14	Skip expression conditionals	559
19.15	Using <code>skip</code> expressions and variables	559
19.16	Inserting skips into the output	560
19.17	Viewing <code>skip</code> variables	560
19.18	Constant skips	561
19.19	Scratch skips	561
19.20	Creating and initialising <code>muskip</code> variables	561
19.21	Setting <code>muskip</code> variables	562
19.22	Using <code>muskip</code> expressions and variables	563
19.23	Viewing <code>muskip</code> variables	564
19.24	Constant muskips	564
19.25	Scratch muskips	564

20	l3keys Implementation	564
20.1	Low-level interface	564
20.2	Constants and variables	568
20.3	The key defining mechanism	570
20.4	Turning properties into actions	572
20.5	Creating key properties	577
20.6	Setting keys	581
20.7	Utilities	587
20.8	Messages	588
21	l3fp implementation	589
22	l3fp-aux implementation	589
22.1	Access to primitives	589
22.2	Internal representation	590
22.3	Using arguments and semicolons	591
22.4	Constants, and structure of floating points	592
22.5	Overflow, underflow, and exact zero	594
22.6	Expanding after a floating point number	594
22.7	Other floating point types	595
22.8	Packing digits	598
22.9	Decimate (dividing by a power of 10)	601
22.10	Functions for use within primitive conditional branches	603
22.11	Integer floating points	604
22.12	Small integer floating points	605
22.13	x-like expansion expandably	605
22.14	Fast string comparison	606
22.15	Name of a function from its l3fp-parse name	606
22.16	Messages	607
23	l3fp-traps Implementation	607
23.1	Flags	607
23.2	Traps	608
23.3	Errors	611
23.4	Messages	611
24	l3fp-round implementation	612
24.1	Rounding tools	613
24.2	The round function	617

25	l3fp-parse implementation	620
25.1	Work plan	620
25.1.1	Storing results	621
25.1.2	Precedence and infix operators	622
25.1.3	Prefix operators, parentheses, and functions	625
25.1.4	Numbers and reading tokens one by one	626
25.2	Main auxiliary functions	628
25.3	Helpers	629
25.4	Parsing one number	630
25.4.1	Numbers: trimming leading zeros	636
25.4.2	Number: small significand	637
25.4.3	Number: large significand	639
25.4.4	Number: beyond 16 digits, rounding	641
25.4.5	Number: finding the exponent	644
25.5	Constants, functions and prefix operators	647
25.5.1	Prefix operators	647
25.5.2	Constants	650
25.5.3	Functions	651
25.6	Main functions	652
25.7	Infix operators	654
25.7.1	Closing parentheses and commas	655
25.7.2	Usual infix operators	657
25.7.3	Juxtaposition	657
25.7.4	Multi-character cases	658
25.7.5	Ternary operator	658
25.7.6	Comparisons	659
25.8	Tools for functions	661
25.9	Candidate: defining new l3fp functions	663
25.10	Messages	665
26	l3fp-assign implementation	665
26.1	Assigning values	666
26.2	Updating values	667
26.3	Showing values	667
26.4	Some useful constants and scratch variables	667
27	l3fp-logic Implementation	668
27.1	Syntax of internal functions	668
27.2	Existence test	668
27.3	Comparison	669
27.4	Floating point expression loops	672
27.5	Extrema	675
27.6	Boolean operations	677
27.7	Ternary operator	678

28	l3fp-basics Implementation	679
28.1	Addition and subtraction	679
28.1.1	Sign, exponent, and special numbers	680
28.1.2	Absolute addition	682
28.1.3	Absolute subtraction	684
28.2	Multiplication	688
28.2.1	Signs, and special numbers	688
28.2.2	Absolute multiplication	689
28.3	Division	692
28.3.1	Signs, and special numbers	692
28.3.2	Work plan	693
28.3.3	Implementing the significand division	695
28.4	Square root	700
28.5	About the sign	707
28.6	Operations on tuples	708
29	l3fp-extended implementation	709
29.1	Description of fixed point numbers	709
29.2	Helpers for numbers with extended precision	710
29.3	Multiplying a fixed point number by a short one	711
29.4	Dividing a fixed point number by a small integer	711
29.5	Adding and subtracting fixed points	713
29.6	Multiplying fixed points	713
29.7	Combining product and sum of fixed points	715
29.8	Extended-precision floating point numbers	717
29.9	Dividing extended-precision numbers	719
29.10	Inverse square root of extended precision numbers	722
29.11	Converting from fixed point to floating point	724
30	l3fp-expo implementation	726
30.1	Logarithm	727
30.1.1	Work plan	727
30.1.2	Some constants	727
30.1.3	Sign, exponent, and special numbers	728
30.1.4	Absolute ln	728
30.2	Exponential	735
30.2.1	Sign, exponent, and special numbers	735
30.3	Power	739

31	l3fp-trig Implementation	746
31.1	Direct trigonometric functions	746
31.1.1	Filtering special cases	747
31.1.2	Distinguishing small and large arguments	750
31.1.3	Small arguments	750
31.1.4	Argument reduction in degrees	751
31.1.5	Argument reduction in radians	752
31.1.6	Computing the power series	758
31.2	Inverse trigonometric functions	761
31.2.1	Arctangent and arccotangent	762
31.2.2	Arcsine and arccosine	766
31.2.3	Arccosecant and arcsecant	768
32	l3fp-convert implementation	770
32.1	Dealing with tuples	770
32.2	Trimming trailing zeros	770
32.3	Scientific notation	771
32.4	Decimal representation	772
32.5	Token list representation	774
32.6	Formatting	775
32.7	Convert to dimension or integer	775
32.8	Convert from a dimension	776
32.9	Use and eval	777
32.10	Convert an array of floating points to a comma list	778
33	l3fp-random Implementation	778
33.1	Engine support	779
33.2	Random floating point	780
33.3	Random integer	781
34	l3intarray implementation	784
34.1	Allocating arrays	784
34.2	Array items	785
35	l3sort implementation	786
35.1	Variables	786
35.2	Finding available \toks registers	787
35.3	Protected user commands	789
35.4	Merge sort	791
35.5	Expandable sorting	794
35.6	Messages	799
35.7	Deprecated functions	801

36	l3tl-analysis implementation	801
36.1	Internal functions	801
36.2	Internal format	801
36.3	Variables and helper functions	802
36.4	Plan of attack	803
36.5	Disabling active characters	805
36.6	First pass	805
36.7	Second pass	810
36.8	Mapping through the analysis	813
36.9	Showing the results	814
36.10	Messages	816
36.11	Deprecated functions	816
37	l3regex implementation	817
37.1	Plan of attack	817
37.2	Helpers	818
37.2.1	Constants and variables	820
37.2.2	Testing characters	821
37.2.3	Character property tests	824
37.2.4	Simple character escape	826
37.3	Compiling	831
37.3.1	Variables used when compiling	832
37.3.2	Generic helpers used when compiling	834
37.3.3	Mode	834
37.3.4	Framework	837
37.3.5	Quantifiers	840
37.3.6	Raw characters	842
37.3.7	Character properties	844
37.3.8	Anchoring and simple assertions	845
37.3.9	Character classes	846
37.3.10	Groups and alternations	849
37.3.11	Catcodes and csnames	852
37.3.12	Raw token lists with \u	855
37.3.13	Other	858
37.3.14	Showing regexes	858
37.4	Building	862
37.4.1	Variables used while building	862
37.4.2	Framework	862
37.4.3	Helpers for building an NFA	864
37.4.4	Building classes	865
37.4.5	Building groups	867
37.4.6	Others	872
37.5	Matching	873
37.5.1	Variables used when matching	874
37.5.2	Matching: framework	876
37.5.3	Using states of the NFA	879
37.5.4	Actions when matching	880
37.6	Replacement	882
37.6.1	Variables and helpers used in replacement	882
37.6.2	Query and brace balance	884

37.6.3	Framework	885
37.6.4	Submatches	887
37.6.5	Csnames in replacement	889
37.6.6	Characters in replacement	890
37.6.7	An error	894
37.7	User functions	894
37.7.1	Variables and helpers for user functions	896
37.7.2	Matching	897
37.7.3	Extracting submatches	897
37.7.4	Replacement	901
37.7.5	Storing and showing compiled patterns	903
37.8	Messages	903
37.9	Code for tracing	909
38	l3box implementation	910
38.1	Support code	910
38.2	Creating and initialising boxes	910
38.3	Measuring and setting box dimensions	911
38.4	Using boxes	912
38.5	Box conditionals	912
38.6	The last box inserted	912
38.7	Constant boxes	913
38.8	Scratch boxes	913
38.9	Viewing box contents	913
38.10	Horizontal mode boxes	914
38.11	Vertical mode boxes	916
38.12	Affine transformations	919
38.13	Deprecated functions	927
39	l3coffins Implementation	928
39.1	Coffins: data structures and general variables	928
39.2	Basic coffin functions	929
39.3	Measuring coffins	933
39.4	Coffins: handle and pole management	934
39.5	Coffins: calculation of pole intersections	936
39.6	Aligning and typesetting of coffins	939
39.7	Coffin diagnostics	944
39.8	Messages	949
40	l3color-base Implementation	950
41	l3luatex implementation	951
41.1	Breaking out to Lua	951
41.2	Messages	952
41.3	Lua functions for internal use	953
41.4	Generic Lua and font support	956
42	l3unicode implementation	956

43	l3candidates Implementation	959
43.1	Additions to l3basics	959
43.2	Additions to l3box	959
43.2.1	Viewing part of a box	959
43.3	Additions to l3clist	962
43.4	Additions to l3coffins	962
43.4.1	Rotating coffins	962
43.4.2	Resizing coffins	967
43.5	Additions to l3file	969
43.6	Additions to l3flag	971
43.7	Additions to l3msg	971
43.8	Additions to l3prg	973
43.9	Additions to l3prop	973
43.10	Additions to l3seq	974
43.11	Additions to l3skip	977
43.12	Additions to l3sys	978
43.13	Additions to l3tl	979
43.13.1	Unicode case changing	982
43.13.2	Building a token list	1004
43.13.3	Other additions to l3tl	1007
43.14	Additions to l3token	1010
44	l3drivers Implementation	1012
44.1	Color support	1013
44.1.1	dvips-style	1013
44.1.2	pdfmode	1014
44.2	dvips driver	1016
44.2.1	Basics	1016
44.2.2	Box operations	1017
44.3	Images	1019
44.4	Drawing	1019
44.5	pdfmode driver	1025
44.5.1	Basics	1025
44.5.2	Box operations	1026
44.6	Images	1027
44.7	dvipdfmx driver	1029
44.7.1	Basics	1029
44.7.2	Box operations	1030
44.8	Images	1031
44.9	xdvipdfmx driver	1033
44.10	Images	1033
44.11	Drawing commands: pdfmode and (x)dvipdfmx	1034
44.12	Drawing	1034
44.13	dvisvgm driver	1041
44.13.1	Basics	1041
44.14	Driver-specific auxiliaries	1041
44.14.1	Box operations	1041
44.15	Images	1043
44.16	Drawing	1044

45	l3deprecation implementation	1050
	Index	1053

Part I

Introduction to expl3 and this document

This document is intended to act as a comprehensive reference manual for the expl3 language. A general guide to the L^AT_EX3 programming language is found in [expl3.pdf](#).

1 Naming functions and variables

L^AT_EX3 does not use `@` as a “letter” for defining internal macros. Instead, the symbols `_` and `:` are used in internal macro names to provide structure. The name of each *function* is divided into logical units using `_`, while `:` separates the *name* of the function from the *argument specifier* (“arg-spec”). This describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The complete list of arg-spec letters for a function is referred to as the *signature* of the function.

Each function name starts with the *module* to which it belongs. Thus apart from a small number of very basic functions, all expl3 function names contain at least one underscore to divide the module name from the descriptive name of the function. For example, all functions concerned with comma lists are in module `clist` and begin `\clist_`.

Every function must include an argument specifier. For functions which take no arguments, this will be blank and the function name will end `:`. Most functions take one or more arguments, and use the following argument specifiers:

- D** The **D** specifier means *do not use*. All of the T_EX primitives are initially `\let` to a **D** name, and some are then given a second name. Only the kernel team should use anything with a **D** specifier!
- N and n** These mean *no manipulation*, of a single token for **N** and of a set of tokens given in braces for **n**. Both pass the argument through exactly as given. Usually, if you use a single token for an **n** argument, all will be well.
- c** This means *csname*, and indicates that the argument will be turned into a csname before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`.
- V and v** These mean *value of variable*. The **V** and **v** specifiers are used to get the content of a variable without needing to worry about the underlying T_EX structure containing the data. A **V** argument will be a single token (similar to **N**), for example `\foo:V \MyVariable`; on the other hand, using **v** a csname is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.
- o** This means *expansion once*. In general, the **V** and **v** specifiers are favoured over **o** for recovering stored information. However, **o** is useful for correctly processing information with delimited arguments.
- x** The **x** specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The T_EX `\edef` primitive carries out this type of expansion. Functions which feature an **x**-type argument are in general *not* expandable, unless specifically noted.

- f** The **f** specifier stands for *full expansion*, and in contrast to **x** stops at the first non-expandable item (reading the argument from left to right) without trying to expand it. For example, when setting a token list variable (a macro used for storage), the sequence

```
\tl_set:Nn \l_my_a_tl { A }
\tl_set:Nn \l_my_b_tl { B }
\tl_set:Nf \l_my_a_tl { \l_my_a_tl \l_my_b_tl }
```

will leave `\l_my_a_tl` with the content `A\l_my_b_tl`, as `A` cannot be expanded and so terminates expansion before `\l_my_b_tl` is considered.

- T and F** For logic tests, there are the branch specifiers **T** (*true*) and **F** (*false*). Both specifiers treat the input in the same way as **n** (no change), but make the logic much easier to see.

- p** The letter **p** indicates *TeX parameters*. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.

- w** Finally, there is the **w** specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some arbitrary string).

Notice that the argument specifier describes how the argument is processed prior to being passed to the underlying function. For example, `\foo:c` will take its argument, convert it to a control sequence and pass it to `\foo:N`.

Variables are named in a similar manner to functions, but begin with a single letter to define the type of variable:

- c** Constant: global parameters whose value should not be changed.

- g** Parameters whose value should only be set globally.

- l** Parameters whose value should only be set locally.

Each variable name is then build up in a similar way to that of a function, typically starting with the module¹ name and then a descriptive part. Variables end with a short identifier to show the variable type:

bool Either true or false.

box Box register.

clist Comma separated list.

coffin a “box with handles” — a higher-level data type for carrying out **box** alignment operations.

dim “Rigid” lengths.

fp floating-point values;

¹The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the `int` module contains some scratch variables called `\l_tmpa_int`, `\l_tmpb_int`, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in `\l_int_tmpa_int` would be very unreadable.

int Integer-valued count register.

prop Property list.

seq “Sequence”: a data-type used to implement lists (with access at both ends) and stacks.

skip “Rubber” lengths.

stream An input or output stream (for reading from or writing to, respectively).

tl Token list variables: placeholder for a token list.

1.1 Terminological inexactitude

A word of warning. In this document, and others referring to the `expl3` programming modules, we often refer to “variables” and “functions” as if they were actual constructs from a real programming language. In truth, `TeX` is a macro processor, and functions are simply macros that may or may not take arguments and expand to their replacement text. Many of the common variables are *also* macros, and if placed into the input stream will simply expand to their definition as well — a “function” with no arguments and a “token list variable” are in truth one and the same. On the other hand, some “variables” are actually registers that must be initialised and their values set and retrieved with specific functions.

The conventions of the `expl3` code are designed to clearly separate the ideas of “macros that contain data” and “macros that contain code”, and a consistent wrapper is applied to all forms of “data” whether they be macros or actually registers. This means that sometimes we will use phrases like “the function returns a value”, when actually we just mean “the macro expands to something”. Similarly, the term “execute” might be used in place of “expand” or it might refer to the more specific case of “processing in `TeX`’s stomach” (if you are familiar with the `TeXbook` parlance).

If in doubt, please ask; chances are we’ve been hasty in writing certain definitions and need to be told to tighten up our terminology.

2 Documentation conventions

This document is typeset with the experimental `l3doc` class; several conventions are used to help describe the features of the code. A number of conventions are used here to make the documentation clearer.

Each group of related functions is given in a box. For a function with a “user” name, this might read:

```
\ExplSyntaxOn
\ExplSyntaxOff
```

```
\ExplSyntaxOn ... \ExplSyntaxOff
```

The textual description of how the function works would appear here. The syntax of the function is shown in mono-spaced text to the right of the box. In this example, the function takes no arguments and so the name of the function is simply reprinted.

For programming functions, which use `_` and `:` in their name there are a few additional conventions: If two related functions are given with identical names but different argument specifiers, these are termed *variants* of each other, and the latter functions are printed in grey to show this more clearly. They will carry out the same function but will take different types of argument:

<code>\seq_new:N</code>	<code>\seq_new:N</code> $\langle sequence \rangle$
<code>\seq_new:c</code>	

When a number of variants are described, the arguments are usually illustrated only for the base function. Here, $\langle sequence \rangle$ indicates that `\seq_new:N` expects the name of a sequence. From the argument specifier, `\seq_new:c` also expects a sequence name, but as a name rather than as a control sequence. Each argument given in the illustration should be described in the following text.

Fully expandable functions Some functions are fully expandable, which allows them to be used within an **x**-type argument (in plain \TeX terms, inside an `\edef`), as well as within an **f**-type argument. These fully expandable functions are indicated in the documentation by a star:

<code>\cs_to_str:N</code> ★	<code>\cs_to_str:N</code> $\langle cs \rangle$
-----------------------------	--

As with other functions, some text should follow which explains how the function works. Usually, only the star will indicate that the function is expandable. In this case, the function expects a $\langle cs \rangle$, shorthand for a $\langle control\ sequence \rangle$.

Restricted expandable functions A few functions are fully expandable but cannot be fully expanded within an **f**-type argument. In this case a hollow star is used to indicate this:

<code>\seq_map_function:NN</code> ☆	<code>\seq_map_function:NN</code> $\langle seq \rangle$ $\langle function \rangle$
-------------------------------------	--

Conditional functions Conditional (**if**) functions are normally defined in three variants, with **T**, **F** and **TF** argument specifiers. This allows them to be used for different “true”/“false” branches, depending on which outcome the conditional is being used to test. To indicate this without repetition, this information is given in a shortened form:

<code>\sys_if_engine_xetex:</code> <u><i>TF</i></u> ★	<code>\sys_if_engine_xetex:TF</code> $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
---	---

The underlining and italic of **TF** indicates that three functions are available:

- `\sys_if_engine_xetex:T`
- `\sys_if_engine_xetex:F`
- `\sys_if_engine_xetex:TF`

Usually, the illustration will use the **TF** variant, and so both $\langle true\ code \rangle$ and $\langle false\ code \rangle$ will be shown. The two variant forms **T** and **F** take only $\langle true\ code \rangle$ and $\langle false\ code \rangle$, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the `expl3` modules should be defined in this way.

Variables, constants and so on are described in a similar manner:

<code>\l_tmpa_tl</code>	
-------------------------	--

A short piece of text will describe the variable: there is no syntax illustration in this case.

In some cases, the function is similar to one in $\text{\LaTeX 2}_{\epsilon}$ or plain \TeX . In these cases, the text will include an extra “ **\TeX hackers note**” section:

<code>\token_to_str:N</code> ★	<code>\token_to_str:N</code> $\langle token \rangle$
--------------------------------	--

The normal description text.

T_EXhackers note: Detail for the experienced T_EX or L^AT_EX 2_ε programmer. In this case, it would point out that this function is the T_EX primitive `\string`.

Changes to behaviour When new functions are added to `expl3`, the date of first inclusion is given in the documentation. Where the documented behaviour of a function changes after it is first introduced, the date of the update will also be given. This means that the programmer can be sure that any release of `expl3` after the date given will contain the function of interest with expected behaviour as described. Note that changes to code internals, including bug fixes, are not recorded in this way *unless* they impact on the expected behaviour.

3 Formal language conventions which apply generally

As this is a formal reference guide for L^AT_EX3 programming, the descriptions of functions are intended to be reasonably “complete”. However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

For tests which have a **TF** argument specification, the test is evaluated to give a logically **TRUE** or **FALSE** result. Depending on this result, either the $\langle true\ code \rangle$ or the $\langle false\ code \rangle$ will be left in the input stream. In the case where the test is expandable, and a predicate (`_p`) variant is available, the logical value determined by the test is left in the input stream: this will typically be part of a larger logical construct.

4 T_EX concepts not supported by L^AT_EX3

The T_EX concept of an “`\outer`” macro is *not supported* at all by L^AT_EX3. As such, the functions provided here may break when used on top of L^AT_EX 2_ε if `\outer` tokens are used in the arguments.

Part II

The l3bootstrap package

Bootstrap code

1 Using the L^AT_EX3 modules

The modules documented in `source3` are designed to be used on top of L^AT_EX 2_ε and are loaded all as one with the usual `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions. These modules will also form the basis of the L^AT_EX3 format, but work in this area is incomplete and not included in this documentation at present.

As the modules use a coding syntax different from standard L^AT_EX 2_ε it provides a few functions for setting it up.

`\ExplSyntaxOn`
`\ExplSyntaxOff`
 Updated: 2011-08-13

`\ExplSyntaxOn` *<code>* `\ExplSyntaxOff`

The `\ExplSyntaxOn` function switches to a category code régime in which spaces are ignored and in which the colon (:) and underscore (_) are treated as “letters”, thus allowing access to the names of code functions and variables. Within this environment, ~ is used to input a space. The `\ExplSyntaxOff` reverts to the document category code régime.

`\ProvidesExplPackage`
`\ProvidesExplClass`
`\ProvidesExplFile`
 Updated: 2017-03-19

`\RequirePackage{expl3}`
`\ProvidesExplPackage` *<package>* *<date>* *<version>* *<description>*

These functions act broadly in the same way as the corresponding L^AT_EX 2_ε kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as L^AT_EX 2_ε provides in turning on `\makeatletter` within package and class code.) The *<date>* should be given in the format *<year>/<month>/<day>*. If the *<version>* is given then it will be prefixed with v in the package identifier line.

`\GetIdInfo`
 Updated: 2012-06-04

`\RequirePackage{l3bootstrap}`
`\GetIdInfo` \$Id: *<SVN info field>* \$ *<description>*

Extracts all information from a SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with `\ExplFileName` for the part of the file name leading up to the period, `\ExplFileDate` for date, `\ExplFileVersion` for version and `\ExplFileDescription` for the description.

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with `\RequirePackage` or similar are loaded with usual L^AT_EX 2_ε category codes and the L^AT_EX3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the `\GetIdInfo` command you can use the information when loading a package with

```
\ProvidesExplPackage{\ExplFileName}
  {\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

Part III

The l3names package

Namespace for primitives

1 Setting up the L^AT_EX3 programming language

This module is at the core of the L^AT_EX3 programming language. It performs the following tasks:

- defines new names for all T_EX primitives;
- switches to the category code régime for programming;
- provides support settings for building the code as a T_EX format.

This module is entirely dedicated to primitives, which should not be used directly within L^AT_EX3 code (outside of “kernel-level” code). As such, the primitives are not documented here: *The T_EXbook*, *T_EX by Topic* and the manuals for pdfT_EX, X_YT_EX and LuaT_EX should be consulted for details of the primitives. These are named based on the engine which first introduced them:

`\tex_...` Introduced by T_EX itself;
`\etex_...` Introduced by the ϵ -T_EX extensions;
`\pdftex_...` Introduced by pdfT_EX;
`\xetex_...` Introduced by X_YT_EX;
`\luatex_...` Introduced by LuaT_EX;
`\utex_...` Introduced by X_YT_EX and LuaT_EX;
`\ptex_...` Introduced by pT_EX;
`\uptex_...` Introduced by upT_EX.

Part IV

The l3basics package

Basic definitions

As the name suggest this package holds some basic definitions which are needed by most or all other packages in this set.

Here we describe those functions that are used all over the place. With that we mean functions dealing with the construction and testing of control sequences. Furthermore the basic parts of conditional processing are covered; conditional processing dealing with specific data types is described in the modules specific for the respective data types.

1 No operation functions

`\prg_do_nothing:` ★**`\prg_do_nothing:`**

An expandable function which does nothing at all: leaves nothing in the input stream after a single expansion.

`\scan_stop:`**`\scan_stop:`**

A non-expandable function which does nothing. Does not vanish on expansion but produces no typeset output.

2 Grouping material

`\group_begin:`
`\group_end:`**`\group_begin:`**
`\group_end:`

These functions begin and end a group for definition purposes. Assignments are local to groups unless carried out in a global manner. (A small number of exceptions to this rule will be noted as necessary elsewhere in this document.) Each **`\group_begin:`** must be matched by a **`\group_end:`**, although this does not have to occur within the same function. Indeed, it is often necessary to start a group within one function and finish it within another, for example when seeking to use non-standard category codes.

`\group_insert_after:N`**`\group_insert_after:N`** *<token>*

Adds *<token>* to the list of *<tokens>* to be inserted when the current group level ends. The list of *<tokens>* to be inserted is empty at the beginning of a group: multiple applications of **`\group_insert_after:N`** may be used to build the inserted list one *<token>* at a time. The current group level may be closed by a **`\group_end:`** function or by a token with category code 2 (close-group), namely a `}` if standard category codes apply.

3 Control sequences and functions

As \TeX is a macro language, creating new functions means creating macros. At point of use, a function is replaced by the replacement text (“code”) in which each parameter in the code (`#1`, `#2`, *etc.*) is replaced the appropriate arguments absorbed by the function. In the following, *code* is therefore used as a shorthand for “replacement text”.

Functions which are not “protected” are fully expanded inside an `x` expansion. In contrast, “protected” functions are not expanded within `x` expansions.

3.1 Defining functions

Functions can be created with no requirement that they are declared first (in contrast to variables, which must always be declared). Declaring a function before setting up the code means that the name chosen is checked and an error raised if it is already in use. The name of a function can be checked at the point of definition using the `\cs_new...` functions: this is recommended for all functions which are defined for the first time.

There are three ways to define new functions. All classes define a function to expand to the substitution text. Within the substitution text the actual parameters are substituted for the formal parameters (`#1`, `#2`, ...).

new Create a new function with the `new` scope, such as `\cs_new:Npn`. The definition is global and results in an error if it is already defined.

set Create a new function with the `set` scope, such as `\cs_set:Npn`. The definition is restricted to the current \TeX group and does not result in an error if the function is already defined.

gset Create a new function with the `gset` scope, such as `\cs_gset:Npn`. The definition is global and does not result in an error if the function is already defined.

Within each set of scope there are different ways to define a function. The differences depend on restrictions on the actual parameters and the expandability of the resulting function.

nopar Create a new function with the `nopar` restriction, such as `\cs_set_nopar:Npn`. The parameter may not contain `\par` tokens.

protected Create a new function with the `protected` restriction, such as `\cs_set_protected:Npn`. The parameter may contain `\par` tokens but the function will not expand within an `x`-type expansion.

Finally, the functions in Subsections 3.2 and 3.3 are primarily meant to define *base functions* only. Base functions can only have the following argument specifiers:

N and n No manipulation.

T and F Functionally equivalent to `n` (you are actually encouraged to use the family of `\prg_new_conditional:` functions described in Section 1).

p and w These are special cases.

The `\cs_new:` functions below (and friends) do not stop you from using other argument specifiers in your function names, but they do not handle expansion for you. You should define the base function and then use `\cs_generate_variant:Nn` to generate custom variants as described in Section 2.

3.2 Defining new functions using parameter text

<code>\cs_new:Npn</code>	<code>\cs_new:Npn <function> <parameters> {<code>}</code>
<code>\cs_new:cpn</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_new:cpx</code>	definition is global and an error results if the <code><function></code> is already defined.

<code>\cs_new_nopar:Npn</code>	<code>\cs_new_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_nopar:cpn</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new_nopar:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. When the
<code>\cs_new_nopar:cpx</code>	<code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The definition

is global and an error results if the `<function>` is already defined.

<code>\cs_new_protected:Npn</code>	<code>\cs_new_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_protected:cpn</code>	Creates <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_new_protected:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_new_protected:cpx</code>	<code><function></code> will not expand within an x-type argument. The definition is global and an

error results if the `<function>` is already defined.

<code>\cs_new_protected_nopar:Npn</code>	<code>\cs_new_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_new_protected_nopar:cpn</code>	
<code>\cs_new_protected_nopar:Npx</code>	
<code>\cs_new_protected_nopar:cpx</code>	

Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The `<function>` will not expand within an x-type argument. The definition is global and an error results if the `<function>` is already defined.

<code>\cs_set:Npn</code>	<code>\cs_set:Npn <function> <parameters> {<code>}</code>
<code>\cs_set:cpn</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_set:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_set:cpx</code>	assignment of a meaning to the <code><function></code> is restricted to the current \TeX group level.

<code>\cs_set_nopar:Npn</code>	<code>\cs_set_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_nopar:cpn</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_set_nopar:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. When the
<code>\cs_set_nopar:cpx</code>	<code><function></code> is used the <code><parameters></code> absorbed cannot contain <code>\par</code> tokens. The assignment

of a meaning to the `<function>` is restricted to the current \TeX group level.

<code>\cs_set_protected:Npn</code>	<code>\cs_set_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_protected:cpn</code>	Sets <code><function></code> to expand to <code><code></code> as replacement text. Within the <code><code></code> , the
<code>\cs_set_protected:Npx</code>	<code><parameters></code> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_set_protected:cpx</code>	assignment of a meaning to the <code><function></code> is restricted to the current \TeX group level.

The `<function>` will not expand within an x-type argument.

<code>\cs_set_protected_nopar:Npn</code>	<code>\cs_set_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_set_protected_nopar:cpn</code>	
<code>\cs_set_protected_nopar:Npx</code>	
<code>\cs_set_protected_nopar:cpx</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level. The $\langle function \rangle$ will not expand within an x -type argument.

<code>\cs_gset:Npn</code>	<code>\cs_gset:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset:cpn</code>	
<code>\cs_gset:Npx</code>	
<code>\cs_gset:cpx</code>	

Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global.

<code>\cs_gset_nopar:Npn</code>	<code>\cs_gset_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_nopar:cpn</code>	
<code>\cs_gset_nopar:Npx</code>	
<code>\cs_gset_nopar:cpx</code>	

Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global.

<code>\cs_gset_protected:Npn</code>	<code>\cs_gset_protected:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_protected:cpn</code>	
<code>\cs_gset_protected:Npx</code>	
<code>\cs_gset_protected:cpx</code>	

Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global. The $\langle function \rangle$ will not expand within an x -type argument.

<code>\cs_gset_protected_nopar:Npn</code>	<code>\cs_gset_protected_nopar:Npn <function> <parameters> {<code>}</code>
<code>\cs_gset_protected_nopar:cpn</code>	
<code>\cs_gset_protected_nopar:Npx</code>	
<code>\cs_gset_protected_nopar:cpx</code>	

Globally sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current \TeX group level: the assignment is global. The $\langle function \rangle$ will not expand within an x -type argument.

3.3 Defining new functions using the signature

<code>\cs_new:Nn</code>	<code>\cs_new:Nn <function> {<code>}</code>
<code>\cs_new:(cn Nx cx)</code>	

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The definition is global and an error results if the $\langle function \rangle$ is already defined.

`\cs_new_nopar:Nn`
`\cs_new_nopar:(cn|Nx|cx)`

`\cs_new_nopar:Nn <function> {<code>}`

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The definition is global and an error results if the $\langle function \rangle$ is already defined.

`\cs_new_protected:Nn`
`\cs_new_protected:(cn|Nx|cx)`

`\cs_new_protected:Nn <function> {<code>}`

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The $\langle function \rangle$ will not expand within an x-type argument. The definition is global and an error results if the $\langle function \rangle$ is already defined.

`\cs_new_protected_nopar:Nn`
`\cs_new_protected_nopar:(cn|Nx|cx)`

`\cs_new_protected_nopar:Nn <function> {<code>}`

Creates $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The $\langle function \rangle$ will not expand within an x-type argument. The definition is global and an error results if the $\langle function \rangle$ is already defined.

`\cs_set:Nn`
`\cs_set:(cn|Nx|cx)`

`\cs_set:Nn <function> {<code>}`

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level.

`\cs_set_nopar:Nn`
`\cs_set_nopar:(cn|Nx|cx)`

`\cs_set_nopar:Nn <function> {<code>}`

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level.

`\cs_set_protected:Nn`
`\cs_set_protected:(cn|Nx|cx)`

`\cs_set_protected:Nn <function> {<code>}`

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, etc.) will be replaced by those absorbed by the function. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current \TeX group level.

<code>\cs_set_protected_nopar:Nn</code>	<code>\cs_set_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_set_protected_nopar:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current TeX group level.

<code>\cs_gset:Nn</code>	<code>\cs_gset:Nn <function> {<code>}</code>
<code>\cs_gset:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_gset_nopar:Nn</code>	<code>\cs_gset_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_nopar:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_gset_protected:Nn</code>	<code>\cs_gset_protected:Nn <function> {<code>}</code>
<code>\cs_gset_protected:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_gset_protected_nopar:Nn</code>	<code>\cs_gset_protected_nopar:Nn <function> {<code>}</code>
<code>\cs_gset_protected_nopar:(cn Nx cx)</code>	

Sets $\langle function \rangle$ to expand to $\langle code \rangle$ as replacement text. Within the $\langle code \rangle$, the number of $\langle parameters \rangle$ is detected automatically from the function signature. These $\langle parameters \rangle$ ($\#1$, $\#2$, *etc.*) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain `\par` tokens. The $\langle function \rangle$ will not expand within an x-type argument. The assignment of a meaning to the $\langle function \rangle$ is global.

<code>\cs_generate_from_arg_count:NNnn</code>	<code>\cs_generate_from_arg_count:NNnn <function> <creator> {<number>}</code>
<code>\cs_generate_from_arg_count:(cNnn Ncnn)</code>	<code>{<code>}</code>

Updated: 2012-01-14

Uses the $\langle creator \rangle$ function (which should have signature Npn , for example `\cs_new:Npn`) to define a $\langle function \rangle$ which takes $\langle number \rangle$ arguments and has $\langle code \rangle$ as replacement text. The $\langle number \rangle$ of arguments is an integer expression, evaluated as detailed for `\int_eval:n`.

3.4 Copying control sequences

Control sequences (not just functions as defined above) can be set to have the same meaning using the functions described here. Making two control sequences equivalent means that the second control sequence is a *copy* of the first (rather than a pointer to it). Thus the old and new control sequence are not tied together: changes to one are not reflected in the other.

In the following text “cs” is used as an abbreviation for “control sequence”.

```
\cs_new_eq:NN
\cs_new_eq:(Nc|cN|cc)
```

```
\cs_new_eq:NN <cs1> <cs2>
\cs_new_eq:NN <cs1> <token>
```

Globally creates $\langle control\ sequence_1 \rangle$ and sets it to have the same meaning as $\langle control\ sequence_2 \rangle$ or $\langle token \rangle$. The second control sequence may subsequently be altered without affecting the copy.

```
\cs_set_eq:NN
\cs_set_eq:(Nc|cN|cc)
```

```
\cs_set_eq:NN <cs1> <cs2>
\cs_set_eq:NN <cs1> <token>
```

Sets $\langle control\ sequence_1 \rangle$ to have the same meaning as $\langle control\ sequence_2 \rangle$ (or $\langle token \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle control\ sequence_1 \rangle$ is restricted to the current \TeX group level.

```
\cs_gset_eq:NN
\cs_gset_eq:(Nc|cN|cc)
```

```
\cs_gset_eq:NN <cs1> <cs2>
\cs_gset_eq:NN <cs1> <token>
```

Globally sets $\langle control\ sequence_1 \rangle$ to have the same meaning as $\langle control\ sequence_2 \rangle$ (or $\langle token \rangle$). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the $\langle control\ sequence_1 \rangle$ is *not* restricted to the current \TeX group level: the assignment is global.

3.5 Deleting control sequences

There are occasions where control sequences need to be deleted. This is handled in a very simple manner.

```
\cs_undefine:N
\cs_undefine:c
```

Updated: 2011-09-15

```
\cs_undefine:N <control\ sequence>
```

Sets $\langle control\ sequence \rangle$ to be globally undefined.

3.6 Showing control sequences

```
\cs_meaning:N ★
\cs_meaning:c ★
```

Updated: 2011-12-22

```
\cs_meaning:N <control\ sequence>
```

This function expands to the *meaning* of the $\langle control\ sequence \rangle$ control sequence. For a macro, this includes the $\langle replacement\ text \rangle$.

\TeX hackers note: This is \TeX ’s $\backslash meaning$ primitive. For tokens that are not control sequences, it is more logical to use $\backslash token_to_meaning:N$. The *c* variant correctly reports undefined arguments.

`\cs_show:N`
`\cs_show:c`

Updated: 2017-02-14

`\cs_show:N` $\langle control\ sequence \rangle$
Displays the definition of the $\langle control\ sequence \rangle$ on the terminal.

T_EXhackers note: This is similar to the T_EX primitive `\show`, wrapped to a fixed number of characters per line.

`\cs_log:N`
`\cs_log:c`

New: 2014-08-22
Updated: 2017-02-14

`\cs_log:N` $\langle control\ sequence \rangle$
Writes the definition of the $\langle control\ sequence \rangle$ in the log file. See also `\cs_show:N` which displays the result in the terminal.

3.7 Converting to and from control sequences

`\use:c` ★

`\use:c` $\{\langle control\ sequence\ name \rangle\}$
Expands the $\langle control\ sequence\ name \rangle$ until only characters remain, and then converts this into a control sequence. This process requires two expansions. As in other `c`-type arguments the $\langle control\ sequence\ name \rangle$ must, when fully expanded, consist of character tokens, typically a mixture of category code 10 (space), 11 (letter) and 12 (other).

T_EXhackers note: Protected macros that appear in a `c`-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

As an example of the `\use:c` function, both

`\use:c { a b c }`

and

`\tl_new:N \l_my_tl`
`\tl_set:Nn \l_my_tl { a b c }`
`\use:c { \tl_use:N \l_my_tl }`

would be equivalent to

`\abc`

after two expansions of `\use:c`.

`\cs_if_exist_use:N` ★
`\cs_if_exist_use:c` ★
`\cs_if_exist_use:NTF` ★
`\cs_if_exist_use:cTF` ★

New: 2012-11-10

`\cs_if_exist_use:N` $\langle control\ sequence \rangle$
`\cs_if_exist_use:NTF` $\langle control\ sequence \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
Tests whether the $\langle control\ sequence \rangle$ is currently defined according to the conditional `\cs_if_exist:NTF` (whether as a function or another control sequence type), and if it is inserts the $\langle control\ sequence \rangle$ into the input stream followed by the $\langle true\ code \rangle$. Otherwise the $\langle false\ code \rangle$ is used.

<code>\cs:w</code>	★	<code>\cs:w</code> <i><control sequence name></i> <code>\cs_end:</code>
<code>\cs_end:</code>	★	

Converts the given *<control sequence name>* into a single control sequence token. This process requires one expansion. The content for *<control sequence name>* may be literal material or from other expandable functions. The *<control sequence name>* must, when fully expanded, consist of character tokens which are not active: typically of category code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

T_EXhackers note: These are the T_EX primitives `\csname` and `\endcsname`.

As an example of the `\cs:w` and `\cs_end:` functions, both

`\cs:w a b c \cs_end:`

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\cs:w \tl_use:N \l_my_tl \cs_end:
```

would be equivalent to

`\abc`

after one expansion of `\cs:w`.

<code>\cs_to_str:N</code>	★	<code>\cs_to_str:N</code> <i><control sequence></i>
---------------------------	---	---

Converts the given *<control sequence>* into a series of characters with category code 12 (other), except spaces, of category code 10. The result does *not* include the current escape token, contrarily to `\token_to_str:N`. Full expansion of this function requires exactly 2 expansion steps, and so an *x*-type expansion, or two *o*-type expansions are required to convert the *<control sequence>* to a sequence of characters in the input stream. In most cases, an *f*-expansion is correct as well, but this loses a space at the start of the result.

4 Analysing control sequence names

<code>\cs_split_function:N</code>	★	<code>\cs_split_function:N</code> <i><function></i>
-----------------------------------	---	---

New: 2018-04-06

Splits the *<function>* into the *<name>* (*i.e.* the part before the colon) and the *<signature>* (*i.e.* after the colon). This information is then placed in the input stream in three parts: the *<name>*, the *<signature>* and a logic token indicating if a colon was found (to differentiate variables from function names). The *<name>* does not include the escape character, and both the *<name>* and *<signature>* are made up of tokens with category code 12 (other).

5 Using or removing tokens and arguments

Tokens in the input can be read and used or read and discarded. If one or more tokens are wrapped in braces then when absorbing them the outer set is removed. At the same time, the category code of each token is set when the token is read by a function (if it is read more than once, the category code is determined by the situation in force when first function absorbs the token).

<hr/>	<hr/>		
<code>\use:n</code>	★	<code>\use:n</code>	$\{\langle group_1 \rangle\}$
<code>\use:nn</code>	★	<code>\use:nn</code>	$\{\langle group_1 \rangle\} \{\langle group_2 \rangle\}$
<code>\use:nnn</code>	★	<code>\use:nnn</code>	$\{\langle group_1 \rangle\} \{\langle group_2 \rangle\} \{\langle group_3 \rangle\}$
<code>\use:nnnn</code>	★	<code>\use:nnnn</code>	$\{\langle group_1 \rangle\} \{\langle group_2 \rangle\} \{\langle group_3 \rangle\} \{\langle group_4 \rangle\}$
<hr/>	<hr/>		

As illustrated, these functions absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument are removed and the remaining tokens are left in the input stream. The category code of these tokens is also fixed by this process (if it has not already been by some other absorption). All of these functions require only a single expansion to operate, so that one expansion of

`\use:nn { abc } { { def } }`

results in the input stream containing

`abc { def }`

i.e. only the outer braces are removed.

<hr/>	<hr/>		
<code>\use_i:nn</code>	★	<code>\use_i:nn</code>	$\{\langle arg_1 \rangle\} \{\langle arg_2 \rangle\}$
<code>\use_ii:nn</code>	★		
<hr/>	<hr/>		

These functions absorb two arguments from the input stream. The function `\use_i:nn` discards the second argument, and leaves the content of the first argument in the input stream. `\use_ii:nn` discards the first argument and leaves the content of the second argument in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

<hr/>	<hr/>		
<code>\use_i:nnn</code>	★	<code>\use_i:nnn</code>	$\{\langle arg_1 \rangle\} \{\langle arg_2 \rangle\} \{\langle arg_3 \rangle\}$
<code>\use_ii:nnn</code>	★		
<code>\use_iii:nnn</code>	★		
<hr/>	<hr/>		

These functions absorb three arguments from the input stream. The function `\use_i:nnn` discards the second and third arguments, and leaves the content of the first argument in the input stream. `\use_ii:nnn` and `\use_iii:nnn` work similarly, leaving the content of second or third arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

<hr/>	<hr/>		
<code>\use_i:nnnn</code>	★	<code>\use_i:nnnn</code>	$\{\langle arg_1 \rangle\} \{\langle arg_2 \rangle\} \{\langle arg_3 \rangle\} \{\langle arg_4 \rangle\}$
<code>\use_ii:nnnn</code>	★		
<code>\use_iii:nnnn</code>	★		
<code>\use_iv:nnnn</code>	★		
<hr/>	<hr/>		

These functions absorb four arguments from the input stream. The function `\use_i:nnnn` discards the second, third and fourth arguments, and leaves the content of the first argument in the input stream. `\use_ii:nnnn`, `\use_iii:nnnn` and `\use_iv:nnnn` work similarly, leaving the content of second, third or fourth arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

<code>\use_i_ii:nnn</code>	★	<code>\use_i_ii:nnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩}</code>
----------------------------	---	--

This function absorbs three arguments and leaves the content of the first and second in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect. An example:

`\use_i_ii:nnn { abc } { { def } } { ghi }`

results in the input stream containing

`abc { def }`

i.e. the outer braces are removed and the third group is removed.

<code>\use_none:n</code>	★	<code>\use_none:n {⟨group₁⟩}</code>
<code>\use_none:nn</code>	★	
<code>\use_none:nnn</code>	★	
<code>\use_none:nnnn</code>	★	
<code>\use_none:nnnnn</code>	★	
<code>\use_none:nnnnnn</code>	★	
<code>\use_none:nnnnnnn</code>	★	
<code>\use_none:nnnnnnnn</code>	★	
<code>\use_none:nnnnnnnnn</code>	★	

These functions absorb between one and nine groups from the input stream, leaving nothing on the resulting input stream. These functions work after a single expansion. One or more of the `n` arguments may be an unbraced single token (*i.e.* an `N` argument).

<code>\use:x</code>		<code>\use:x {⟨expandable tokens⟩}</code>
---------------------	--	---

Updated: 2011-12-31 Fully expands the `⟨expandable tokens⟩` and inserts the result into the input stream at the current location. Any hash characters (`#`) in the argument must be doubled.

5.1 Selecting tokens from delimited arguments

A different kind of function for selecting tokens from the token stream are those that use delimited arguments.

<code>\use_none_delimit_by_q_nil:w</code>	★	<code>\use_none_delimit_by_q_nil:w ⟨balanced text⟩ \q_nil</code>
<code>\use_none_delimit_by_q_stop:w</code>	★	<code>\use_none_delimit_by_q_stop:w ⟨balanced text⟩ \q_stop</code>
<code>\use_none_delimit_by_q_recursion_stop:w</code>	★	<code>\use_none_delimit_by_q_recursion_stop:w ⟨balanced text⟩ \q_recursion_stop</code>

Absorb the `⟨balanced text⟩` from the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

<code>\use_i_delimit_by_q_nil:nw</code>	★	<code>\use_i_delimit_by_q_nil:nw {⟨inserted tokens⟩} ⟨balanced text⟩</code>
<code>\use_i_delimit_by_q_stop:nw</code>	★	<code>\q_nil</code>
<code>\use_i_delimit_by_q_recursion_stop:nw</code>	★	<code>\use_i_delimit_by_q_stop:nw {⟨inserted tokens⟩} ⟨balanced text⟩ \q_stop</code>
		<code>\use_i_delimit_by_q_recursion_stop:nw {⟨inserted tokens⟩} ⟨balanced text⟩ \q_recursion_stop</code>

Absorb the `⟨balanced text⟩` from the input stream delimited by the marker given in the function name, leaving `⟨inserted tokens⟩` in the input stream for further processing.

6 Predicates and conditionals

L^AT_EX3 has three concepts for conditional flow processing:

Branching conditionals Functions that carry out a test and then execute, depending on its result, either the code supplied as the *⟨true code⟩* or the *⟨false code⟩*. These arguments are denoted with T and F, respectively. An example would be

```
\cs_if_free:cTF {abc} {⟨true code⟩} {⟨false code⟩}
```

a function that turns the first argument into a control sequence (since it’s marked as c) then checks whether this control sequence is still free and then depending on the result carries out the code in the second argument (true case) or in the third argument (false case).

These type of functions are known as “conditionals”; whenever a TF function is defined it is usually accompanied by T and F functions as well. These are provided for convenience when the branch only needs to go a single way. Package writers are free to choose which types to define but the kernel definitions always provide all three versions.

Important to note is that these branching conditionals with *⟨true code⟩* and/or *⟨false code⟩* are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream.

These conditional functions may or may not be fully expandable, but if they are expandable they are accompanied by a “predicate” for the same test as described below.

Predicates “Predicates” are functions that return a special type of boolean value which can be tested by the boolean expression parser. All functions of this type are expandable and have names that end with `_p` in the description part. For example,

```
\cs_if_free_p:N
```

would be a predicate function for the same type of test as the conditional described above. It would return “true” if its argument (a single token denoted by N) is still free for definition. It would be used in constructions like

```
\bool_if:nTF {
  \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl
} {⟨true code⟩} {⟨false code⟩}
```

For each predicate defined, a “branching conditional” also exists that behaves like a conditional described above.

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain T_EX and L^AT_EX 2_ε. Their use is discouraged in `expl3` (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

```
\c_true_bool
\c_false_bool
```

Constants that represent `true` and `false`, respectively. Used to implement predicates.

6.1 Tests on control sequences

<code>\cs_if_eq_p:NN</code>	★	<code>\cs_if_eq_p:NN <cs₁> <cs₂></code>
<code>\cs_if_eq:NNTF</code>	★	<code>\cs_if_eq:NNTF <cs₁> <cs₂> {\true code} {\false code}</code>

Compares the definition of two *<control sequences>* and is logically **true** if they are the same, *i.e.* if they have exactly the same definition when examined with `\cs_show:N`.

<code>\cs_if_exist_p:N</code>	★	<code>\cs_if_exist_p:N <control sequence></code>
<code>\cs_if_exist_p:c</code>	★	<code>\cs_if_exist:NNTF <control sequence> {\true code} {\false code}</code>
<code>\cs_if_exist:NTF</code>	★	Tests whether the <i><control sequence></i> is currently defined (whether as a function or another control sequence type). Any definition of <i><control sequence></i> other than <code>\relax</code> evaluates as true .
<code>\cs_if_exist:cTF</code>	★	

<code>\cs_if_free_p:N</code>	★	<code>\cs_if_free_p:N <control sequence></code>
<code>\cs_if_free_p:c</code>	★	<code>\cs_if_free:NNTF <control sequence> {\true code} {\false code}</code>
<code>\cs_if_free:NTF</code>	★	Tests whether the <i><control sequence></i> is currently free to be defined. This test is false if the <i><control sequence></i> currently exists (as defined by <code>\cs_if_exist:N</code>).
<code>\cs_if_free:cTF</code>	★	

6.2 Primitive conditionals

The ε -TeX engine itself provides many different conditionals. Some expand whatever comes after them and others don't. Hence the names for these underlying functions often contains a `:w` part but higher level functions are often available. See for instance `\int_compare_p:nNn` which is a wrapper for `\if_int_compare:w`.

Certain conditionals deal with specific data types like boxes and fonts and are described there. The ones described below are either the universal conditionals or deal with control sequences. We prefix primitive conditionals with `\if_`.

<code>\if_true:</code>	★	<code>\if_true: <true code> \else: <false code> \fi:</code>
<code>\if_false:</code>	★	<code>\if_false: <true code> \else: <false code> \fi:</code>
<code>\else:</code>	★	<code>\reverse_if:N <primitive conditional></code>
<code>\fi:</code>	★	<code>\if_true:</code> always executes <i><true code></i> , while <code>\if_false:</code> always executes <i><false code></i> . <code>\reverse_if:N</code> reverses any two-way primitive conditional. <code>\else:</code> and <code>\fi:</code> delimit the branches of the conditional. The function <code>\or:</code> is documented in <code>l3int</code> and used in case switches.
<code>\reverse_if:N</code>	★	

T_EXhackers note: These are equivalent to their corresponding T_EX primitive conditionals; `\reverse_if:N` is ε -T_EX's `\unless`.

<code>\if_meaning:w</code>	★	<code>\if_meaning:w <arg₁> <arg₂> <true code> \else: <false code> \fi:</code>
----------------------------	---	---

`\if_meaning:w` executes *<true code>* when *<arg₁>* and *<arg₂>* are the same, otherwise it executes *<false code>*. *<arg₁>* and *<arg₂>* could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

T_EXhackers note: This is T_EX's `\ifx`.

<code>\if:w</code>	★	<code>\if:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_charcode:w</code>	★	<code>\if_catcode:w <token₁> <token₂> <true code> \else: <false code> \fi:</code>
<code>\if_catcode:w</code>	★	These conditionals expand any following tokens until two unexpandable tokens are left. If you wish to prevent this expansion, prefix the token in question with <code>\exp_not:N</code> . <code>\if_catcode:w</code> tests if the category codes of the two tokens are the same whereas <code>\if:w</code> tests if the character codes are identical. <code>\if_charcode:w</code> is an alternative name for <code>\if:w</code> .

<code>\if_cs_exist:N</code>	★	<code>\if_cs_exist:N <cs> <true code> \else: <false code> \fi:</code>
<code>\if_cs_exist:w</code>	★	<code>\if_cs_exist:w <tokens> \cs_end: <true code> \else: <false code> \fi:</code>

Check if `<cs>` appears in the hash table or if the control sequence that can be formed from `<tokens>` appears in the hash table. The latter function does not turn the control sequence in question into `\scan_stop:!` This can be useful when dealing with control sequences which cannot be entered as a single token.

<code>\if_mode_horizontal:</code>	★	<code>\if_mode_horizontal: <true code> \else: <false code> \fi:</code>
<code>\if_mode_vertical:</code>	★	Execute <code><true code></code> if currently in horizontal mode, otherwise execute <code><false code></code> . Similar for the other functions.
<code>\if_mode_math:</code>	★	
<code>\if_mode_inner:</code>	★	

Part V

The l3expan package

Argument expansion

This module provides generic methods for expanding T_EX arguments in a systematic manner. The functions in this module all have prefix `exp`.

Not all possible variations are implemented for every base function. Instead only those that are used within the L^AT_EX3 kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

1 Defining new variants

The definition of variant forms for base functions may be necessary when writing new functions or when applying a kernel function in a situation that we haven't thought of before.

Internally preprocessing of arguments is done with functions from the `\exp_` module. They all look alike, an example would be `\exp_args:NNo`. This function has three arguments, the first and the second are a single tokens, while the third argument should be given in braces. Applying `\exp_args:NNo` expands the content of third argument once before any expansion of the first and second arguments. If `\seq_gpush:No` was not defined it could be coded in the following way:

```
\exp_args:NNo \seq_gpush:Nn
  \g_file_name_stack
  { \l_tmpa_tl }
```

In other words, the first argument to `\exp_args:NNo` is the base function and the other arguments are preprocessed and then passed to this base function. In the example the first argument to the base function should be a single token which is left unchanged while the second argument is expanded once. From this example we can also see how the variants are defined. They just expand into the appropriate `\exp_` function followed by the desired base function, *e.g.*

```
\cs_generate_variant:Nn \seq_gpush:Nn { No }
```

results in the definition of `\seq_gpush:No`

```
\cs_new:Npn \seq_gpush:No { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is safe as the `\cs_generate_variant:Nn` function will only create new definitions if there is not already one available. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

The steps above may be automated by using the function `\cs_generate_variant:Nn`, described next.

2 Methods for defining variants

We recall the set of available argument specifiers.

- `N` is used for single-token arguments while `c` constructs a control sequence from its name and passes it to a parent function as an `N`-type argument.
- Many argument types extract or expand some tokens and provide it as an `n`-type argument, namely a braced multiple-token argument: `V` extracts the value of a variable, `v` extracts the value from the name of a variable, `n` uses the argument as it is, `o` expands once, `f` expands fully the first token, `x` expands fully all tokens at the price of being non-expandable.
- A few odd argument types remain: `T` and `F` for conditional processing, otherwise identical to `n`-type arguments, `p` for the parameter text in definitions, `w` for arguments with a specific syntax, and `D` to denote primitives that should not be used directly.

`\cs_generate_variant:Nn`
`\cs_generate_variant:cn`

Updated: 2017-11-28

`\cs_generate_variant:Nn` \langle parent control sequence \rangle $\{$ \langle variant argument specifiers \rangle $\}$

This function is used to define argument-specifier variants of the \langle parent control sequence \rangle for L^AT_EX3 code-level macros. The \langle parent control sequence \rangle is first separated into the \langle base name \rangle and \langle original argument specifier \rangle . The comma-separated list of \langle variant argument specifiers \rangle is then used to define variants of the \langle original argument specifier \rangle if these are not already defined. For each \langle variant \rangle given, a function is created that expands its arguments as detailed and passes them to the \langle parent control sequence \rangle . So for example

```
\cs_set:Npn \foo:Nn #1#2 { code here }
\cs_generate_variant:Nn \foo:Nn { c }
```

creates a new function `\foo:cn` which expands its first argument into a control sequence name and passes the result to `\foo:Nn`. Similarly

```
\cs_generate_variant:Nn \foo:Nn { NV , cV }
```

generates the functions `\foo:NV` and `\foo:cV` in the same way. The `\cs_generate_variant:Nn` function can only be applied if the \langle parent control sequence \rangle is already defined. If the \langle parent control sequence \rangle is protected or if the \langle variant \rangle involves x arguments, then the \langle variant control sequence \rangle is also protected. The \langle variant \rangle is created globally, as is any `\exp_args:N` \langle variant \rangle function needed to carry out the expansion.

Only n and N arguments can be changed to other types. The only allowed changes are

- c variant of an N parent;
- o , V , v , f , or x variant of an n parent;
- N , n , T , F , or p argument unchanged.

This means the \langle parent \rangle of a \langle variant \rangle form is always unambiguous, even in cases where both an n -type parent and an N -type parent exist, such as for `\tl_count:n` and `\tl_count:N`.

For backward compatibility it is currently possible to make n , o , V , v , f , or x -type variants of an N -type argument or N or c -type variants of an n -type argument. Both are deprecated. The first because passing more than one token to an N -type argument will typically break the parent function's code. The second because programmers who use that most often want to access the value of a variable given its name, hence should use a V -type or v -type variant instead of c -type. In those cases, using the lower-level `\exp_args:No` or `\exp_args:Nc` functions explicitly is preferred to defining confusing variants.

3 Introducing the variants

It is usually best to follow the following guidelines when defining new functions that are supposed to come with variant forms.

- Arguments that need full expansion (*i.e.*, are denoted with x) should be avoided if possible as they can not be processed expandably, *i.e.*, functions of this type cannot work correctly in arguments that are themselves subject to x expansion.

When speed is essential (for functions that do very little work and are used numerous times in a document) the following applies because internal functions for argument expansion come in two flavours, some faster than others.

- Arguments that might need expansion should come first in the list of arguments.
- Arguments that should consist of single tokens `N`, `c`, `V`, or `v` should come first among these.
- Arguments that appear after the first multi-token argument `n`, `f` or `o` require slightly slower special processing to be expanded. Therefore it is best to use the optimized functions, namely those that contain only `N`, `c`, `V`, and `v`, and, in the last position, `o`, `f`, with possible trailing `N` or `n` or `T` or `F`, which are not expanded.

The `V` type returns the value of a register, which can be one of `tl`, `clist`, `int`, `skip`, `dim`, `muskip`, or built-in `TEX` registers. The `v` type is the same except it first creates a control sequence out of its argument before returning the value.

In general, the programmer should not need to be concerned with expansion control. When simply using the content of a variable, functions with a `V` specifier should be used. For those referred to by `(cs)name`, the `v` specifier is available for the same purpose. Only when specific expansion steps are needed, such as when using delimited arguments, should the lower-level functions with `o` specifiers be employed.

The `f` type is so special that it deserves an example. It is typically used in contexts where only expandable commands are allowed. Then `x`-expansion cannot be used, and `f`-expansion provides an alternative that expands as much as can be done in such contexts. For instance, say that we want to evaluate the integer expression `3+4` and pass the result `7` as an argument to an expandable function `\example:n`. For this, one should define a variant using `\cs_generate_variant:Nn \example:n { f }`, then do

```
\example:f { \int_eval:n { 3 + 4 } }
```

Note that `x`-expansion would also expand `\int_eval:n` fully to its result `7`, but the variant `\example:x` cannot be expandable. Note also that `o`-expansion would not expand `\int_eval:n` fully to its result since that function requires several expansions. Besides the fact that `x`-expansion is protected rather than expandable, another difference between `f`-expansion and `x`-expansion is that `f`-expansion expands tokens from the beginning and stops as soon as a non-expandable token is encountered, while `x`-expansion continues expanding further tokens. Thus, for instance

```
\example:f { \int_eval:n { 1 + 2 } , \int_eval:n { 3 + 4 } }
```

results in the call

```
\example:n { 3 , \int_eval:n { 3 + 4 } }
```

while using `\example:x` instead results in

```
\example:n { 3 , 7 }
```

at the cost of being protected. If you use this type of expansion in conditional processing then you should stick to using `TF` type functions only as the expansion does not finish any `\if... \fi`: itself!

It is important to note that both `f`- and `o`-type expansion are concerned with the expansion of tokens from left to right in their arguments. In particular, `o`-type expansion applies to the first *token* in the argument it receives: it is conceptually similar to

`\exp_after:wN <base function> \exp_after:wN { <argument> }`

At the same time, `f`-type expansion stops at the *first* non-expandable token. This means for example that both

`\tl_set:No \l_tmpa_tl { { \g_tmpb_tl } }`

and

`\tl_set:Nf \l_tmpa_tl { { \g_tmpb_tl } }`

leave `\g_tmpb_tl` unchanged: `{` is the first token in the argument and is non-expandable.

4 Manipulating the first argument

These functions are described in detail: expansion of multiple tokens follows the same rules but is described in a shorter fashion.

<code>\exp_args:Nc</code>	★	<code>\exp_args:Nc <function> {<tokens>}</code>
<code>\exp_args:cc</code>	★	

This function absorbs two arguments (the `<function>` name and the `<tokens>`). The `<tokens>` are expanded until only characters remain, and are then turned into a control sequence. The result is inserted into the input stream *after* reinsertion of the `<function>`. Thus the `<function>` may take more than one argument: all others are left unchanged.

The `:cc` variant constructs the `<function>` name in the same manner as described for the `<tokens>`.

T_EXhackers note: Protected macros that appear in a `c`-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

<code>\exp_args:No</code>	★	<code>\exp_args:No <function> {<tokens>} ...</code>
---------------------------	---	---

This function absorbs two arguments (the `<function>` name and the `<tokens>`). The `<tokens>` are expanded once, and the result is inserted in braces into the input stream *after* reinsertion of the `<function>`. Thus the `<function>` may take more than one argument: all others are left unchanged.

<code>\exp_args:NV</code>	★	<code>\exp_args:NV <function> <variable></code>
---------------------------	---	---

This function absorbs two arguments (the names of the `<function>` and the `<variable>`). The content of the `<variable>` are recovered and placed inside braces into the input stream *after* reinsertion of the `<function>`. Thus the `<function>` may take more than one argument: all others are left unchanged.

`\exp_args:Nv` ★ `\exp_args:Nv <function> {<tokens>}`

This function absorbs two arguments (the *<function>* name and the *<tokens>*). The *<tokens>* are expanded until only characters remain, and are then turned into a control sequence. This control sequence should be the name of a *<variable>*. The content of the *<variable>* are recovered and placed inside braces into the input stream *after* reinsertion of the *<function>*. Thus the *<function>* may take more than one argument: all others are left unchanged.

TeXhackers note: Protected macros that appear in a v-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

`\exp_args:Nf` ★ `\exp_args:Nf <function> {<tokens>}`

This function absorbs two arguments (the *<function>* name and the *<tokens>*). The *<tokens>* are fully expanded until the first non-expandable token is found (if that is a space it is removed), and the result is inserted in braces into the input stream *after* reinsertion of the *<function>*. Thus the *<function>* may take more than one argument: all others are left unchanged.

`\exp_args:Nx` `\exp_args:Nx <function> {<tokens>}`

This function absorbs two arguments (the *<function>* name and the *<tokens>*) and exhaustively expands the *<tokens>*. The result is inserted in braces into the input stream *after* reinsertion of the *<function>*. Thus the *<function>* may take more than one argument: all others are left unchanged.

5 Manipulating two arguments

`\exp_args:NNc` ★ `\exp_args:NNc <token12`

`\exp_args:NNo` ★
`\exp_args:NNV` ★
`\exp_args:NNv` ★
`\exp_args:NNf` ★

These optimized functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments.

`\exp_args:Ncc` ★
`\exp_args:Nco` ★
`\exp_args:NcV` ★
`\exp_args:Ncv` ★
`\exp_args:Ncf` ★
`\exp_args:NVV` ★

<code>\exp_args:Nnc</code>	★	<code>\exp_args:Noo</code>	$\langle token \rangle \{ \langle tokens_1 \rangle \} \{ \langle tokens_2 \rangle \}$
<code>\exp_args:Nno</code>	★	These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions need slower processing.	
<code>\exp_args:NnV</code>	★		
<code>\exp_args:Nnv</code>	★		
<code>\exp_args:Nnf</code>	★		
<code>\exp_args:Noc</code>	★		
<code>\exp_args:Noo</code>	★	These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions are not expandable due to their x-type argument.	
<code>\exp_args:Nof</code>	★		
<code>\exp_args:NVo</code>	★		
<code>\exp_args:Nfo</code>	★		
<code>\exp_args:Nff</code>	★		

Updated: 2012-01-14

<code>\exp_args:NNx</code>		<code>\exp_args:NNx</code>	$\langle token_1 \rangle \langle token_2 \rangle \{ \langle tokens \rangle \}$
<code>\exp_args:Ncx</code>		These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions are not expandable due to their x-type argument.	
<code>\exp_args:Nnx</code>			
<code>\exp_args:Nox</code>			
<code>\exp_args:Nxo</code>			
<code>\exp_args:Nxx</code>			

6 Manipulating three arguments

<code>\exp_args:NNNo</code>	★	<code>\exp_args:NNNo</code>	$\langle token_1 \rangle \langle token_2 \rangle \langle token_3 \rangle \{ \langle tokens \rangle \}$
<code>\exp_args:NNNV</code>	★	These optimized functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, <i>etc.</i>	
<code>\exp_args:Nccc</code>	★		
<code>\exp_args:NcNc</code>	★		
<code>\exp_args:NcNo</code>	★		
<code>\exp_args:Ncco</code>	★		

<code>\exp_args:NNcf</code>	★	<code>\exp_args:NNoo</code>	$\langle token_1 \rangle \langle token_2 \rangle \{ \langle token_3 \rangle \} \{ \langle tokens \rangle \}$
<code>\exp_args:NNno</code>	★	These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, <i>etc.</i> These functions need slower processing.	
<code>\exp_args:NNnV</code>	★		
<code>\exp_args:NNoo</code>	★		
<code>\exp_args:NNVV</code>	★		
<code>\exp_args:Ncno</code>	★		
<code>\exp_args:NcnV</code>	★		
<code>\exp_args:Ncoo</code>	★		
<code>\exp_args:NcVV</code>	★		
<code>\exp_args:Nnnc</code>	★		
<code>\exp_args:Nnno</code>	★		
<code>\exp_args:Nnnf</code>	★		
<code>\exp_args:Nnff</code>	★		
<code>\exp_args:Nooo</code>	★		
<code>\exp_args:Noof</code>	★		
<code>\exp_args:Nffo</code>	★		

```

\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox

```

New: 2015-08-12

```
\exp_args:NNnx <token_1> <token_2> {\tokens_1} {\tokens_2}
```

These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*

7 Unbraced expansion

```

\exp_last_unbraced:No      ★ \exp_last_unbraced:Nno <token> {\tokens_1} {\tokens_2}
\exp_last_unbraced:(NV|Nv|Nf) ★
\exp_last_unbraced:NNo    ★
\exp_last_unbraced:(NNV|Nco|NcV) ★
\exp_last_unbraced:Nno    ★
\exp_last_unbraced:(Noo|Nfo) ★
\exp_last_unbraced:NNNo   ★
\exp_last_unbraced:NNNV   ★
\exp_last_unbraced:NnNo   ★
\exp_last_unbraced:NNNNo  ★

```

Updated: 2012-02-12

These functions absorb the number of arguments given by their specification, carry out the expansion indicated and leave the results in the input stream, with the last argument not surrounded by the usual braces. Of these, the :Nno, :Noo, :Nfo and :NnNo variants need slower processing.

T_EXhackers note: As an optimization, the last argument is unbraced by some of those functions before expansion. This can cause problems if the argument is empty: for instance, `\exp_last_unbraced:Nf \foo_bar:w { } \q_stop` leads to an infinite loop, as the quark is **f**-expanded.

```
\exp_last_unbraced:Nx \exp_last_unbraced:Nx <function> {\tokens}
```

This function fully expands the `<tokens>` and leaves the result in the input stream after reinsertion of the `<function>`. This function is not expandable.

```
\exp_last_two_unbraced:Noo ★ \exp_last_two_unbraced:Noo <token> {\tokens_1} {\tokens_2}
```

This function absorbs three arguments and expands the second and third once. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments, which are not wrapped in braces. This function needs special (slower) processing.

`\exp_after:wN` ★**`\exp_after:wN`** $\langle token_1 \rangle$ $\langle token_2 \rangle$

Carries out a single expansion of $\langle token_2 \rangle$ (which may consume arguments) prior to the expansion of $\langle token_1 \rangle$. If $\langle token_2 \rangle$ has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that $\langle token_1 \rangle$ may be *any* single token, including group-opening and -closing tokens (`{` or `}` assuming normal \TeX category codes). Unless specifically required this should be avoided: expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_arg:N` function.

\TeX hackers note: This is the \TeX primitive `\expandafter` renamed.

8 Preventing expansion

Despite the fact that the following functions are all about preventing expansion, they're designed to be used in an expandable context and hence are all marked as being 'expandable' since they themselves disappear after the expansion has completed.

`\exp_not:N` ★**`\exp_not:N`** $\langle token \rangle$

Prevents expansion of the $\langle token \rangle$ in a context where it would otherwise be expanded, for example an **x**-type argument or the first token in an **o** or **f** argument.

\TeX hackers note: This is the \TeX `\noexpand` primitive. It only prevents expansion. At the beginning of an **f**-type argument, a space $\langle token \rangle$ is removed even if it appears as `\exp_not:N \c_space_token`. In an **x**-expanding definition (`\cs_new:Npx`), a macro parameter introduces an argument even if it appears as `\exp_not:N #1`. This differs from `\exp_not:n`.

`\exp_not:c` ★**`\exp_not:c`** $\{\langle tokens \rangle\}$

Expands the $\langle tokens \rangle$ until only characters remain, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited using `\exp_not:N`.

\TeX hackers note: Protected macros that appear in a **c**-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

`\exp_not:n` ★**`\exp_not:n`** $\{\langle tokens \rangle\}$

Prevents expansion of the $\langle tokens \rangle$ in an **x**-type argument. In all other cases the $\langle tokens \rangle$ continue to be expanded, for example in the input stream or in other types of arguments such as **c**, **f**, **v**. The argument of `\exp_not:n` *must* be surrounded by braces.

\TeX hackers note: This is the ε - \TeX `\unexpanded` primitive. In an **x**-expanding definition (`\cs_new:Npx`), `\exp_not:n {#1}` is equivalent to `##1` rather than to `#1`, namely it inserts the two characters `#` and `1`.

`\exp_not:o` ★**`\exp_not:o`** $\{\langle tokens \rangle\}$

Expands the $\langle tokens \rangle$ once, then prevents any further expansion in **x**-type arguments using `\exp_not:n`.

<code>\exp_not:V</code>	★	<code>\exp_not:V</code>	<code><variable></code>
-------------------------	---	-------------------------	-------------------------------

Recovers the content of the `<variable>`, then prevents expansion of this material in `x`-type arguments using `\exp_not:n`.

<code>\exp_not:v</code>	★	<code>\exp_not:v</code>	<code>{<tokens>}</code>
-------------------------	---	-------------------------	-------------------------------

Expands the `<tokens>` until only characters remains, and then converts this into a control sequence which should be a `<variable>` name. The content of the `<variable>` is recovered, and further expansion in `x`-type arguments is prevented using `\exp_not:n`.

T_EXhackers note: Protected macros that appear in a `v`-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

<code>\exp_not:f</code>	★	<code>\exp_not:f</code>	<code>{<tokens>}</code>
-------------------------	---	-------------------------	-------------------------------

Expands `<tokens>` fully until the first unexpandable token is found (if it is a space it is removed). Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion in `x`-type arguments using `\exp_not:n`.

<code>\exp_stop_f:</code>	★	<code>\foo_bar:f</code>	<code>{ <tokens> \exp_stop_f: <more tokens> }</code>
---------------------------	---	-------------------------	--

Updated: 2011-06-03

This function terminates an `f`-type expansion. Thus if a function `\foo_bar:f` starts an `f`-type expansion and all of `<tokens>` are expandable `\exp_stop_f:` terminates the expansion of tokens even if `<more tokens>` are also expandable. The function itself is an implicit space token. Inside an `x`-type expansion, it retains its form, but when typeset it produces the underlying space (□).

9 Controlled expansion

The `expl3` language makes all efforts to hide the complexity of T_EX expansion from the programmer by providing concepts that evaluate/expand arguments of functions prior to calling the “base” functions. Thus, instead of using many `\expandafter` calls and other trickery it is usually a matter of choosing the right variant of a function to achieve a desired result.

Of course, deep down T_EX is using expansion as always and there are cases where a programmer needs to control that expansion directly; typical situations are basic data manipulation tools. This section documents the functions for that level. These commands are used throughout the kernel code, but we hope that outside the kernel there will be little need to resort to them. Instead the argument manipulation methods document above should usually be sufficient.

While `\exp_after:wN` expands one token (out of order) it is sometimes necessary to expand several tokens in one go. The next set of commands provide this functionality. Be aware that it is absolutely required that the programmer has full control over the tokens to be expanded, i.e., it is not possible to use these functions to expand unknown input as part of `<expandable-tokens>` as that will break badly if unexpandable tokens are encountered in that place!

`\exp:w` ★
`\exp_end:` ★

New: 2015-08-23

`\exp:w` $\langle expandable\ tokens \rangle$ `\exp_end:`

Expands $\langle expandable-tokens \rangle$ until reaching `\exp_end:` at which point expansion stops. The full expansion of $\langle expandable\ tokens \rangle$ has to be empty. If any token in $\langle expandable\ tokens \rangle$ or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end:` will be misinterpreted later on.²

In typical use cases the `\exp_end:` is hidden somewhere in the replacement text of $\langle expandable-tokens \rangle$ rather than being on the same expansion level than `\exp:w`, e.g., you may see code such as

`\exp:w \@@_case:NnTF #1 {#2} { } { }`

where somewhere during the expansion of `\@@_case:NnTF` the `\exp_end:` gets generated.

T_EXhackers note: The current implementation uses `\romannumeral` hence ignores space tokens and explicit signs + and - in the expansion of the $\langle expandable\ tokens \rangle$, but this should not be relied upon.

`\exp:w` ★
`\exp_end_continue_f:w` ★

New: 2015-08-23

`\exp:w` $\langle expandable-tokens \rangle$ `\exp_end_continue_f:w` $\langle further-tokens \rangle$

Expands $\langle expandable-tokens \rangle$ until reaching `\exp_end_continue_f:w` at which point expansion continues as an f-type expansion expanding $\langle further-tokens \rangle$ until an unexpandable token is encountered (or the f-type expansion is explicitly terminated by `\exp_stop_f:`). As with all f-type expansions a space ending the expansion gets removed.

The full expansion of $\langle expandable-tokens \rangle$ has to be empty. If any token in $\langle expandable-tokens \rangle$ or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end_continue_f:w` will be misinterpreted later on.³

In typical use cases $\langle expandable-tokens \rangle$ contains no tokens at all, e.g., you will see code such as

`\exp_after:wN { \exp:w \exp_end_continue_f:w #2 }`

where the `\exp_after:wN` triggers an f-expansion of the tokens in #2. For technical reasons this has to happen using two tokens (if they would be hidden inside another command `\exp_after:wN` would only expand the command but not trigger any additional f-expansion).

You might wonder why there are two different approaches available, after all the effect of

`\exp:w` $\langle expandable-tokens \rangle$ `\exp_end:`

can be alternatively achieved through an f-type expansion by using `\exp_stop_f:`, i.e.

`\exp:w \exp_end_continue_f:w` $\langle expandable-tokens \rangle$ `\exp_stop_f:`

The reason is simply that the first approach is slightly faster (one less token to parse and less expansion internally) so in places where such performance really matters and where we want to explicitly stop the expansion at a defined point the first form is preferable.

²Due to the implementation you might get the character in position 0 in the current font (typically “”) in the output without any error message!

³In this particular case you may get a character into the output as well as an error message.

<code>\exp:w</code>	★
<code>\exp_end_continue_f:nw</code>	★

New: 2015-08-23

`\exp:w` $\langle expandable-tokens \rangle$ `\exp_end_continue_f:nw` $\langle further-tokens \rangle$

The difference to `\exp_end_continue_f:w` is that we first we pick up an argument which is then returned to the input stream. If $\langle further-tokens \rangle$ starts with space tokens then these space tokens are removed while searching for the argument. If it starts with a brace group then the braces are removed. Thus such spaces or braces will not terminate the f-type expansion.

10 Internal functions

`\::n` `\cs_new:Npn \exp_args:Ncof { \::c \::o \::f \::: }`

`\::N` Internal forms for the base expansion types. These names do *not* conform to the general \LaTeX 3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

`\::p`

`\::c`

`\::o`

`\::f`

`\::x`

`\::v`

`\::V`

`\:::`

`\::o_unbraced` `\cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }`

`\::f_unbraced` Internal forms for the expansion types which leave the terminal argument unbraced. These names do *not* conform to the general \LaTeX 3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

`\::x_unbraced`

`\::v_unbraced`

`\::V_unbraced`

Part VI

The l3tl package

Token lists

T_EX works with tokens, and L^AT_EX3 therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called “token list variable”, which have the suffix `tl`: a token list variable can also be used as the argument to a function, for example

```
\foo:N \l_some_tl
```

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix `tl`. In many cases, functions which can be applied to token list variables are paired with similar functions for application to explicit lists of tokens: the two “views” of a token list are therefore collected together here.

A token list (explicit, or stored in a variable) can be seen either as a list of “items”, or a list of “tokens”. An item is whatever `\use:n` would grab as its argument: a single non-space token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or `␣`, `{`, or `}` (assuming normal T_EX category codes). Thus for example

```
{ Hello } ~ world
```

contains six items (Hello, w, o, r, l and d), but thirteen tokens (`{`, H, e, l, l, o, `}`, `␣`, w, o, r, l and d). Functions which act on items are often faster than their analogue acting directly on tokens.

1 Creating and initialising token list variables

<code>\tl_new:N</code>	<code>\tl_new:N <tl var></code>
<code>\tl_new:c</code>	

Creates a new `<tl var>` or raises an error if the name is already taken. The declaration is global. The `<tl var>` is initially empty.

<code>\tl_const:Nn</code>	<code>\tl_const:Nn <tl var> {<token list>}</code>
<code>\tl_const:(Nx cn cx)</code>	

Creates a new constant `<tl var>` or raises an error if the name is already taken. The value of the `<tl var>` is set globally to the `<token list>`.

<code>\tl_clear:N</code>	<code>\tl_clear:N <tl var></code>
<code>\tl_clear:c</code>	
<code>\tl_gclear:N</code>	
<code>\tl_gclear:c</code>	

Clears all entries from the `<tl var>`.

<hr/>	
<code>\tl_clear_new:N</code>	<code>\tl_clear_new:N <tl var></code>
<code>\tl_clear_new:c</code>	
<code>\tl_gclear_new:N</code>	Ensures that the $\langle tl\ var \rangle$ exists globally by applying <code>\tl_new:N</code> if necessary, then applies
<code>\tl_gclear_new:c</code>	<code>\tl_(g)clear:N</code> to leave the $\langle tl\ var \rangle$ empty.
<hr/>	
<code>\tl_set_eq:NN</code>	<code>\tl_set_eq:NN <tl var₁> <tl var₂></code>
<code>\tl_set_eq:(cN Nc cc)</code>	Sets the content of $\langle tl\ var_1 \rangle$ equal to that of $\langle tl\ var_2 \rangle$.
<code>\tl_gset_eq:NN</code>	
<code>\tl_gset_eq:(cN Nc cc)</code>	
<hr/>	
<code>\tl_concat:NNN</code>	<code>\tl_concat:NNN <tl var₁> <tl var₂> <tl var₃></code>
<code>\tl_concat:ccc</code>	
<code>\tl_gconcat:NNN</code>	Concatenates the content of $\langle tl\ var_2 \rangle$ and $\langle tl\ var_3 \rangle$ together and saves the result in
<code>\tl_gconcat:ccc</code>	$\langle tl\ var_1 \rangle$. The $\langle tl\ var_2 \rangle$ is placed at the left side of the new token list.
<hr/>	
New: 2012-05-18	
<hr/>	
<code>\tl_if_exist_p:N</code> *	<code>\tl_if_exist_p:N <tl var></code>
<code>\tl_if_exist_p:c</code> *	<code>\tl_if_exist:NTF <tl var> {\true code} {\false code}</code>
<code>\tl_if_exist:NTF</code> *	
<code>\tl_if_exist:cTF</code> *	Tests whether the $\langle tl\ var \rangle$ is currently defined. This does not check that the $\langle tl\ var \rangle$ really is a token list variable.
<hr/>	
New: 2012-03-03	

2 Adding data to token list variables

<hr/>	
<code>\tl_set:Nn</code>	<code>\tl_set:Nn <tl var> {\tokens}</code>
<code>\tl_set:(NV Nv No Nf Nx cn cV cv co cf cx)</code>	
<code>\tl_gset:Nn</code>	
<code>\tl_gset:(NV Nv No Nf Nx cn cV cv co cf cx)</code>	
<hr/>	
Sets $\langle tl\ var \rangle$ to contain $\langle tokens \rangle$, removing any previous content from the variable.	
<hr/>	
<code>\tl_put_left:Nn</code>	<code>\tl_put_left:Nn <tl var> {\tokens}</code>
<code>\tl_put_left:(NV No Nx cn cV co cx)</code>	
<code>\tl_gput_left:Nn</code>	
<code>\tl_gput_left:(NV No Nx cn cV co cx)</code>	
<hr/>	
Appends $\langle tokens \rangle$ to the left side of the current content of $\langle tl\ var \rangle$.	
<hr/>	
<code>\tl_put_right:Nn</code>	<code>\tl_put_right:Nn <tl var> {\tokens}</code>
<code>\tl_put_right:(NV No Nx cn cV co cx)</code>	
<code>\tl_gput_right:Nn</code>	
<code>\tl_gput_right:(NV No Nx cn cV co cx)</code>	
<hr/>	
Appends $\langle tokens \rangle$ to the right side of the current content of $\langle tl\ var \rangle$.	

3 Modifying token list variables

```
\tl_replace_once:Nnn
\tl_replace_once:cnn
\tl_greplace_once:Nnn
\tl_greplace_once:cnn
```

Updated: 2011-08-11

```
\tl_replace_once:Nnn <tl var> {{<old tokens>}} {{<new tokens>}}
```

Replaces the first (leftmost) occurrence of *<old tokens>* in the *<tl var>* with *<new tokens>*. *<Old tokens>* cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```
\tl_replace_all:Nnn
\tl_replace_all:cnn
\tl_greplace_all:Nnn
\tl_greplace_all:cnn
```

Updated: 2011-08-11

```
\tl_replace_all:Nnn <tl var> {{<old tokens>}} {{<new tokens>}}
```

Replaces all occurrences of *<old tokens>* in the *<tl var>* with *<new tokens>*. *<Old tokens>* cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern *<old tokens>* may remain after the replacement (see `\tl_remove_all:Nn` for an example).

```
\tl_remove_once:Nn
\tl_remove_once:cn
\tl_gremove_once:Nn
\tl_gremove_once:cn
```

Updated: 2011-08-11

```
\tl_remove_once:Nn <tl var> {{<tokens>}}
```

Removes the first (leftmost) occurrence of *<tokens>* from the *<tl var>*. *<Tokens>* cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

```
\tl_remove_all:Nn
\tl_remove_all:cn
\tl_gremove_all:Nn
\tl_gremove_all:cn
```

Updated: 2011-08-11

```
\tl_remove_all:Nn <tl var> {{<tokens>}}
```

Removes all occurrences of *<tokens>* from the *<tl var>*. *<Tokens>* cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern *<tokens>* may remain after the removal, for instance,

```
\tl_set:Nn \l_tmpa_tl {abbccd} \tl_remove_all:Nn \l_tmpa_tl {bc}
```

results in `\l_tmpa_tl` containing `abcd`.

4 Reassigning token list category codes

These functions allow the rescanning of tokens: re-apply T_EX's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes).

<code>\tl_set_rescan:Nnn</code>	<code>\tl_set_rescan:Nnn <tl var> {<setup>} {<tokens>}</code>
<code>\tl_set_rescan:(Nno Nnx cnn cno cnx)</code>	
<code>\tl_gset_rescan:Nnn</code>	
<code>\tl_gset_rescan:(Nno Nnx cnn cno cnx)</code>	

Updated: 2015-08-11

Sets $\langle tl\ var \rangle$ to contain $\langle tokens \rangle$, applying the category code régime specified in the $\langle setup \rangle$ before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the $\langle setup \rangle$ are those in force at the point of use of `\tl_set_rescan:Nnn`.) This allows the $\langle tl\ var \rangle$ to contain material with category codes other than those that apply when $\langle tokens \rangle$ are absorbed. The $\langle setup \rangle$ is run within a group and may contain any valid input, although only changes in category codes are relevant. See also `\tl_rescan:nn`.

T_EXhackers note: The $\langle tokens \rangle$ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user $\langle setup \rangle$), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

<code>\tl_rescan:nn</code>	<code>\tl_rescan:nn {<setup>} {<tokens>}</code>
----------------------------	---

Updated: 2015-08-11

Rescans $\langle tokens \rangle$ applying the category code régime specified in the $\langle setup \rangle$, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the $\langle setup \rangle$ are those in force at the point of use of `\tl_rescan:nn`.) The $\langle setup \rangle$ is run within a group and may contain any valid input, although only changes in category codes are relevant. See also `\tl_set_rescan:Nnn`, which is more robust than using `\tl_set:Nn` in the $\langle tokens \rangle$ argument of `\tl_rescan:nn`.

T_EXhackers note: The $\langle tokens \rangle$ are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user $\langle setup \rangle$), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

5 Token list conditionals

<code>\tl_if_blank_p:n</code>	★	<code>\tl_if_blank_p:n {<token list>}</code>
<code>\tl_if_blank_p:(V o)</code>	★	<code>\tl_if_blank:nTF {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_blank:nTF</code>	★	
<code>\tl_if_blank:(V o)TF</code>	★	

Tests if the $\langle token\ list \rangle$ consists only of blank spaces (*i.e.* contains no item). The test is **true** if $\langle token\ list \rangle$ is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is **false** otherwise.

<code>\tl_if_empty_p:N</code>	★	<code>\tl_if_empty_p:N <tl var></code>
<code>\tl_if_empty_p:c</code>	★	<code>\tl_if_empty:NNTF <tl var> {<true code>} {<false code>}</code>
<code>\tl_if_empty:nTF</code>	★	Tests if the <i><token list variable></i> is entirely empty (<i>i.e.</i> contains no tokens at all).
<code>\tl_if_empty:cTF</code>	★	

<code>\tl_if_empty_p:n</code>	★	<code>\tl_if_empty_p:n {<token list>}</code>
<code>\tl_if_empty_p:(V o)</code>	★	<code>\tl_if_empty:nNTF {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_empty:nTF</code>	★	Tests if the <i><token list></i> is entirely empty (<i>i.e.</i> contains no tokens at all).
<code>\tl_if_empty:(V o)TF</code>	★	

New: 2012-05-24
Updated: 2012-06-05

<code>\tl_if_eq_p:NN</code>	★	<code>\tl_if_eq_p:NN <tl var₁> <tl var₂></code>
<code>\tl_if_eq_p:(Nc cN cc)</code>	★	<code>\tl_if_eq:NNTF <tl var₁> <tl var₂> {<true code>} {<false code>}</code>
<code>\tl_if_eq:NNTF</code>	★	Compares the content of two <i><token list variables></i> and is logically true if the two contain the same list of tokens (<i>i.e.</i> identical in both the list of characters they contain and the category codes of those characters). Thus for example
<code>\tl_if_eq:(Nc cN cc)TF</code>	★	

```

\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Nx \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:NNTF \l_tmpa_tl \l_tmpb_tl { true } { false }

```

yields **false**.

<code>\tl_if_eq:nnTF</code>	★	<code>\tl_if_eq:nnTF {<token list₁>} {<token list₂>} {<true code>} {<false code>}</code>
-----------------------------	---	--

Tests if *<token list₁>* and *<token list₂>* contain the same list of tokens, both in respect of character codes and category codes.

<code>\tl_if_in:NnTF</code>	★	<code>\tl_if_in:NnTF <tl var> {<token list>} {<true code>} {<false code>}</code>
<code>\tl_if_in:cnTF</code>	★	Tests if the <i><token list></i> is found in the content of the <i><tl var></i> . The <i><token list></i> cannot contain the tokens <code>{</code> , <code>}</code> or <code>#</code> (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_if_in:nnTF</code>	★	<code>\tl_if_in:nnTF {<token list₁>} {<token list₂>} {<true code>} {<false code>}</code>
<code>\tl_if_in:(Vn on no)TF</code>	★	Tests if <i><token list₂></i> is found inside <i><token list₁></i> . The <i><token list₂></i> cannot contain the tokens <code>{</code> , <code>}</code> or <code>#</code> (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_if_novalue_p:n</code>	★	<code>\tl_if_novalue_p:n {<token list>}</code>
<code>\tl_if_novalue:nTF</code>	★	<code>\tl_if_novalue:nTF {<token list>} {<true code>} {<false code>}</code>

New: 2017-11-14

Tests if the *<token list>* is exactly equal to the special `\c_novalue_tl` marker. This function is intended to allow construction of flexible document interface structures in which missing optional arguments are detected.

<code>\tl_if_single_p:N</code> ★	<code>\tl_if_single_p:N <tl var></code>
<code>\tl_if_single_p:c</code> ★	<code>\tl_if_single:NNTF <tl var> {\true code} {\false code}</code>
<code>\tl_if_single:NTF</code> ★	Tests if the content of the <i><tl var></i> consists of a single item, <i>i.e.</i> is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to <code>\tl_count:N</code> .
<code>\tl_if_single:cTF</code> ★	
Updated: 2011-08-13	

<code>\tl_if_single_p:n</code> ★	<code>\tl_if_single_p:n {\token list}</code>
<code>\tl_if_single:nTF</code> ★	<code>\tl_if_single:nNTF {\token list} {\true code} {\false code}</code>
Updated: 2011-08-13	
Tests if the <i><token list></i> has exactly one item, <i>i.e.</i> is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to <code>\tl_count:n</code> .	

<code>\tl_case:Nn</code> ★	<code>\tl_case:NnNTF <test token list variable></code>
<code>\tl_case:cn</code> ★	{
<code>\tl_case:NnTF</code> ★	<i><token list variable case₁></i> {\code case ₁ }
<code>\tl_case:cnTF</code> ★	<i><token list variable case₂></i> {\code case ₂ }
...	
	<i><token list variable case_n></i> {\code case _n }
	}
	{\true code}
	{\false code}
New: 2013-07-24	

This function compares the *<test token list variable>* in turn with each of the *<token list variable cases>*. If the two are equal (as described for `\tl_if_eq:NNTF`) then the associated *<code>* is left in the input stream and other cases are discarded. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The function `\tl_case:Nn`, which does nothing if there is no match, is also available.

6 Mapping to token lists

<code>\tl_map_function:NN</code> ☆	<code>\tl_map_function:NN <tl var> <function></code>
<code>\tl_map_function:cN</code> ☆	Applies <i><function></i> to every <i><item></i> in the <i><tl var></i> . The <i><function></i> receives one argument for each iteration. This may be a number of tokens if the <i><item></i> was stored within braces. Hence the <i><function></i> should anticipate receiving n-type arguments. See also <code>\tl_map_function:nN</code> .
Updated: 2012-06-29	

<code>\tl_map_function:nN</code> ☆	<code>\tl_map_function:nN {\token list} <function></code>
Updated: 2012-06-29	
Applies <i><function></i> to every <i><item></i> in the <i><token list></i> , The <i><function></i> receives one argument for each iteration. This may be a number of tokens if the <i><item></i> was stored within braces. Hence the <i><function></i> should anticipate receiving n-type arguments. See also <code>\tl_map_function:NN</code> .	

<code>\tl_map_inline:Nn</code>	<code>\tl_map_inline:Nn <tl var> {\inline function}</code>
<code>\tl_map_inline:cn</code>	Applies the <i><inline function></i> to every <i><item></i> stored within the <i><tl var></i> . The <i><inline function></i> should consist of code which receives the <i><item></i> as #1. See also <code>\tl_map_function:NN</code> .
Updated: 2012-06-29	

<hr/> <code>\tl_map_inline:nn</code> <hr/>	<code>\tl_map_inline:nn {<token list>} {<inline function>}</code>
Updated: 2012-06-29	Applies the <i><inline function></i> to every <i><item></i> stored within the <i><token list></i> . The <i><inline function></i> should consist of code which receives the <i><item></i> as #1. See also <code>\tl_map_function:nn</code> .
<hr/> <code>\tl_map_variable:NNn</code> <code>\tl_map_variable:cNn</code> <hr/>	<code>\tl_map_variable:NNn <tl var> <variable> {<code>}</code>
Updated: 2012-06-29	Stores each <i><item></i> of the <i><tl var></i> in turn in the (token list) <i><variable></i> and applies the <i><code></i> . The <i><code></i> will usually make use of the <i><variable></i> , but this is not enforced. The assignments to the <i><variable></i> are local. See also <code>\tl_map_inline:Nn</code> .
<hr/> <code>\tl_map_variable:nNn</code> <hr/>	<code>\tl_map_variable:nNn {<token list>} <variable> {<code>}</code>
Updated: 2012-06-29	Stores each <i><item></i> of the <i><token list></i> in turn in the (token list) <i><variable></i> and applies the <i><code></i> . The <i><code></i> will usually make use of the <i><variable></i> , but this is not enforced. The assignments to the <i><variable></i> are local. See also <code>\tl_map_inline:nn</code> .
<hr/> <code>\tl_map_break: ☆</code> <hr/>	<code>\tl_map_break:</code>
Updated: 2012-06-29	Used to terminate a <code>\tl_map...</code> function before all entries in the <i><token list variable></i> have been processed. This normally takes place within a conditional statement, for example <pre> \tl_map_inline:Nn \l_my_tl { \str_if_eq:nnT { #1 } { bingo } { \tl_map_break: } % Do something useful }</pre> <p>See also <code>\tl_map_break:n</code>. Use outside of a <code>\tl_map...</code> scenario leads to low level TeX errors.</p> <p>TeXhackers note: When the mapping is broken, additional tokens may be inserted before the <i><tokens></i> are inserted into the input stream. This depends on the design of the mapping function.</p>

`\tl_map_break:n` ☆

Updated: 2012-06-29

`\tl_map_break:n` {<code>}

Used to terminate a `\tl_map...` function before all entries in the *<token list variable>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
{
  \str_if_eq:nnT { #1 } { bingo }
  { \tl_map_break:n { <code> } }
  % Do something useful
}
```

Use outside of a `\tl_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

7 Using token lists

`\tl_to_str:n` ☆

`\tl_to_str:V` ☆

`\tl_to_str:n` {<token list>}

Converts the *<token list>* to a *<string>*, leaving the resulting character tokens in the input stream. A *<string>* is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This function requires only a single expansion. Its argument *must* be braced.

TeXhackers note: This is the ϵ -TeX primitive `\detokenize`. Converting a *<token list>* to a *<string>* yields a concatenation of the string representations of every token in the *<token list>*. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative or greater than the largest character code;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not “letter”.

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally `#`). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n {\a}` normally produces the three character “backslash”, “lower-case a”, “space”, but it may also produce a single “lower-case a” if the escape character is negative and `a` is currently not a letter.

`\tl_to_str:N` ☆

`\tl_to_str:c` ☆

`\tl_to_str:N` <tl var>

Converts the content of the *<tl var>* into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This *<string>* is then left in the input stream. For low-level details, see the notes given for `\tl_to_str:n`.

<code>\tl_use:N</code>	★	<code>\tl_use:N <tl var></code>
------------------------	---	---------------------------------------

<code>\tl_use:c</code>	★
------------------------	---

Recovers the content of a $\langle tl\ var \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a $\langle tl\ var \rangle$ directly without an accessor function.

8 Working with the content of token lists

<code>\tl_count:n</code>	★	<code>\tl_count:n {\tokens}</code>
--------------------------	---	------------------------------------

<code>\tl_count:(V o)</code>	★
------------------------------	---

New: 2012-05-13

Counts the number of $\langle items \rangle$ in $\langle tokens \rangle$ and leaves this information in the input stream. Unbraced tokens count as one element as do each token group $\{\dots\}$. This process ignores any unprotected spaces within $\langle tokens \rangle$. See also `\tl_count:N`. This function requires three expansions, giving an $\langle integer\ denotation \rangle$.

<code>\tl_count:N</code>	★
--------------------------	---

<code>\tl_count:c</code>	★
--------------------------	---

New: 2012-05-13

`\tl_count:N <tl var>`

Counts the number of token groups in the $\langle tl\ var \rangle$ and leaves this information in the input stream. Unbraced tokens count as one element as do each token group $\{\dots\}$. This process ignores any unprotected spaces within the $\langle tl\ var \rangle$. See also `\tl_count:n`. This function requires three expansions, giving an $\langle integer\ denotation \rangle$.

<code>\tl_reverse:n</code>	★
----------------------------	---

<code>\tl_reverse:(V o)</code>	★
--------------------------------	---

Updated: 2012-01-08

`\tl_reverse:n {\token list}`

Reverses the order of the $\langle items \rangle$ in the $\langle token\ list \rangle$, so that $\langle item_1 \rangle \langle item_2 \rangle \langle item_3 \rangle \dots \langle item_n \rangle$ becomes $\langle item_n \rangle \dots \langle item_3 \rangle \langle item_2 \rangle \langle item_1 \rangle$. This process preserves unprotected space within the $\langle token\ list \rangle$. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider `\tl_reverse_items:n`. See also `\tl_reverse:N`.

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an `x`-type argument expansion.

<code>\tl_reverse:N</code>

<code>\tl_reverse:c</code>

<code>\tl_greverse:N</code>

<code>\tl_greverse:c</code>

Updated: 2012-01-08

`\tl_reverse:N <tl var>`

Reverses the order of the $\langle items \rangle$ stored in $\langle tl\ var \rangle$, so that $\langle item_1 \rangle \langle item_2 \rangle \langle item_3 \rangle \dots \langle item_n \rangle$ becomes $\langle item_n \rangle \dots \langle item_3 \rangle \langle item_2 \rangle \langle item_1 \rangle$. This process preserves unprotected spaces within the $\langle token\ list\ variable \rangle$. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. See also `\tl_reverse:n`, and, for improved performance, `\tl_reverse_items:n`.

<code>\tl_reverse_items:n</code>	★
----------------------------------	---

New: 2012-01-08

`\tl_reverse_items:n {\token list}`

Reverses the order of the $\langle items \rangle$ stored in $\langle tl\ var \rangle$, so that $\{\langle item_1 \rangle\} \{\langle item_2 \rangle\} \{\langle item_3 \rangle\} \dots \{\langle item_n \rangle\}$ becomes $\{\langle item_n \rangle\} \dots \{\langle item_3 \rangle\} \{\langle item_2 \rangle\} \{\langle item_1 \rangle\}$. This process removes any unprotected space within the $\langle token\ list \rangle$. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function `\tl_reverse:n`.

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an `x`-type argument expansion.

<code>\tl_trim_spaces:n</code> ★	<code>\tl_trim_spaces:n {⟨token list⟩}</code>
<code>\tl_trim_spaces:o</code> ★	
New: 2011-07-09	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the <i>⟨token list⟩</i> and leaves the result in the input stream.
Updated: 2012-06-25	

TeXhackers note: The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an *x*-type argument expansion.

<code>\tl_trim_spaces_apply:nN</code> ★	<code>\tl_trim_spaces_apply:nN {⟨token list⟩} ⟨function⟩</code>
<code>\tl_trim_spaces_apply:oN</code> ★	
New: 2018-04-12	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the <i>⟨token list⟩</i> and passes the result to the <i>⟨function⟩</i> as an <i>n</i> -type argument.

<code>\tl_trim_spaces:N</code>	<code>\tl_trim_spaces:N ⟨tl var⟩</code>
<code>\tl_trim_spaces:c</code>	
<code>\tl_gtrim_spaces:N</code>	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the content of the <i>⟨tl var⟩</i> . Note that this therefore <i>resets</i> the content of the variable.
<code>\tl_gtrim_spaces:c</code>	
New: 2011-07-09	

<code>\tl_sort:Nn</code>	<code>\tl_sort:Nn ⟨tl var⟩ {⟨comparison code⟩}</code>
<code>\tl_sort:cn</code>	
<code>\tl_gsort:Nn</code>	Sorts the items in the <i>⟨tl var⟩</i> according to the <i>⟨comparison code⟩</i> , and assigns the result to <i>⟨tl var⟩</i> . The details of sorting comparison are described in Section 1.
<code>\tl_gsort:cn</code>	
New: 2017-02-06	

<code>\tl_sort:nN</code> ★	<code>\tl_sort:nN {⟨token list⟩} ⟨conditional⟩</code>
New: 2017-02-06	Sorts the items in the <i>⟨token list⟩</i> , using the <i>⟨conditional⟩</i> to compare items, and leaves the result in the input stream. The <i>⟨conditional⟩</i> should have signature <code>:nnTF</code> , and return true if the two items being compared should be left in the same order, and false if the items should be swapped. The details of sorting comparison are described in Section 1.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an *x*-type argument expansion.

9 The first token from a token list

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

<code>\tl_head:N</code> ★	<code>\tl_head:n {⟨token list⟩}</code>
<code>\tl_head:n</code> ★	
<code>\tl_head:(V v f)</code> ★	Leaves in the input stream the first <i>⟨item⟩</i> in the <i>⟨token list⟩</i> , discarding the rest of the <i>⟨token list⟩</i> . All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example
Updated: 2012-09-09	

`\tl_head:n { abc }`

and

`\tl_head:n { ~ abc }`

both leave `a` in the input stream. If the “head” is a brace group, rather than a single token, the braces are removed, and so

`\tl_head:n { ~ { ~ ab } c }`

yields `▯ab`. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) results in `\tl_head:n` leaving nothing in the input stream.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an *x*-type argument expansion.

<code>\tl_head:w</code> ★	<code>\tl_head:w ⟨token list⟩ { } \q_stop</code>
---------------------------	--

Leaves in the input stream the first *⟨item⟩* in the *⟨token list⟩*, discarding the rest of the *⟨token list⟩*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank *⟨token list⟩* (which consists only of space characters) results in a low-level TeX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, `\tl_if_blank:nF` may be used to avoid using the function with a “blank” argument. This function requires only a single expansion, and thus is suitable for use within an *o*-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

<code>\tl_tail:N</code> ★	<code>\tl_tail:n {⟨token list⟩}</code>
<code>\tl_tail:n</code> ★	
<code>\tl_tail:(V v f)</code> ★	Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first <i>⟨item⟩</i> in the <i>⟨token list⟩</i> , and leaves the remaining tokens in the input stream. Thus for example
Updated: 2012-09-01	

`\tl_tail:n { a ~ {bc} d }`

and

`\tl_tail:n { ~ a ~ {bc} d }`

both leave `▯{bc}d` in the input stream. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) results in `\tl_tail:n` leaving nothing in the input stream.

TeXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an *x*-type argument expansion.

<code>\tl_if_head_eq_catcode_p:nN</code>	★	<code>\tl_if_head_eq_catcode_p:nN {<token list>} <test token></code>
<code>\tl_if_head_eq_catcode:nNTF</code>	★	<code>\tl_if_head_eq_catcode:nNTF {<token list>} <test token></code>
		<code>{<true code>} {<false code>}</code>

Updated: 2012-07-09

Tests if the first *<token>* in the *<token list>* has the same category code as the *<test token>*. In the case where the *<token list>* is empty, the test is always **false**.

<code>\tl_if_head_eq_charcode_p:nN</code>	★	<code>\tl_if_head_eq_charcode_p:nN {<token list>} <test token></code>
<code>\tl_if_head_eq_charcode_p:fN</code>	★	<code>\tl_if_head_eq_charcode:nNTF {<token list>} <test token></code>
<code>\tl_if_head_eq_charcode:nNTF</code>	★	<code>{<true code>} {<false code>}</code>
<code>\tl_if_head_eq_charcode:fNTF</code>	★	

Updated: 2012-07-09

Tests if the first *<token>* in the *<token list>* has the same character code as the *<test token>*. In the case where the *<token list>* is empty, the test is always **false**.

<code>\tl_if_head_eq_meaning_p:nN</code>	★	<code>\tl_if_head_eq_meaning_p:nN {<token list>} <test token></code>
<code>\tl_if_head_eq_meaning:nNTF</code>	★	<code>\tl_if_head_eq_meaning:nNTF {<token list>} <test token></code>
		<code>{<true code>} {<false code>}</code>

Updated: 2012-07-09

Tests if the first *<token>* in the *<token list>* has the same meaning as the *<test token>*. In the case where *<token list>* is empty, the test is always **false**.

<code>\tl_if_head_is_group_p:n</code>	★	<code>\tl_if_head_is_group_p:n {<token list>}</code>
<code>\tl_if_head_is_group:nTF</code>	★	<code>\tl_if_head_is_group:nTF {<token list>} {<true code>} {<false code>}</code>

New: 2012-07-08

Tests if the first *<token>* in the *<token list>* is an explicit begin-group character (with category code 1 and any character code), in other words, if the *<token list>* starts with a brace group. In particular, the test is **false** if the *<token list>* starts with an implicit token such as `\c_group_begin_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

<code>\tl_if_head_is_N_type_p:n</code>	★	<code>\tl_if_head_is_N_type_p:n {<token list>}</code>
<code>\tl_if_head_is_N_type:nTF</code>	★	<code>\tl_if_head_is_N_type:nTF {<token list>} {<true code>} {<false code>}</code>

New: 2012-07-08

Tests if the first *<token>* in the *<token list>* is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields **false**, as it does not have a “normal” first token. This function is useful to implement actions on token lists on a token by token basis.

<code>\tl_if_head_is_space_p:n</code>	★	<code>\tl_if_head_is_space_p:n {<token list>}</code>
<code>\tl_if_head_is_space:nTF</code>	★	<code>\tl_if_head_is_space:nTF {<token list>} {<true code>} {<false code>}</code>

Updated: 2012-07-08

Tests if the first *<token>* in the *<token list>* is an explicit space character (explicit token with character code 12 and category code 10). In particular, the test is **false** if the *<token list>* starts with an implicit token such as `\c_space_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

10 Using a single item

<hr/> <code>\tl_item:nn</code> ★	<code>\tl_item:nn {(token list)} {(integer expression)}</code>
<code>\tl_item:Nn</code> ★	Indexing items in the $\langle token list \rangle$ from 1 on the left, this function evaluates the $\langle integer expression \rangle$ and leaves the appropriate item from the $\langle token list \rangle$ in the input stream. If the $\langle integer expression \rangle$ is negative, indexing occurs from the right of the token list, starting at -1 for the right-most item. If the index is out of bounds, then the function expands to nothing.
<code>\tl_item:cn</code> ★	
<hr/> New: 2014-07-17 <hr/>	

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an x-type argument expansion.

11 Viewing token lists

<hr/> <code>\tl_show:N</code>	<code>\tl_show:N \tl var</code>
<code>\tl_show:c</code>	Displays the content of the $\langle tl var \rangle$ on the terminal.
<hr/> Updated: 2015-08-01 <hr/>	

TeXhackers note: This is similar to the TeX primitive `\show`, wrapped to a fixed number of characters per line.

<hr/> <code>\tl_show:n</code>	<code>\tl_show:n {(token list)}</code>
<hr/> Updated: 2015-08-07 <hr/>	Displays the $\langle token list \rangle$ on the terminal.

TeXhackers note: This is similar to the ϵ -TeX primitive `\showtokens`, wrapped to a fixed number of characters per line.

<hr/> <code>\tl_log:N</code>	<code>\tl_log:N \tl var</code>
<code>\tl_log:c</code>	Writes the content of the $\langle tl var \rangle$ in the log file. See also <code>\tl_show:N</code> which displays the result in the terminal.
<hr/> New: 2014-08-22 <hr/> Updated: 2015-08-01 <hr/>	

<hr/> <code>\tl_log:n</code>	<code>\tl_log:n {(token list)}</code>
<hr/> New: 2014-08-22 <hr/> Updated: 2015-08-07 <hr/>	Writes the $\langle token list \rangle$ in the log file. See also <code>\tl_show:n</code> which displays the result in the terminal.

12 Constant token lists

<hr/> <code>\c_empty_tl</code> <hr/>	Constant that is always empty.
--------------------------------------	--------------------------------

<hr/> <code>\c_novalue_tl</code> <hr/>	A marker for the absence of an argument. This constant <code>tl</code> can safely be typeset (cf. <code>\q_nil</code>), with the result being <code>-NoValue-</code> . It is important to note that <code>\c_novalue_tl</code> is constructed such that it will <i>not</i> match the simple text input <code>-NoValue-</code> , <i>i.e.</i> that
--	---

`\tl_if_eq:VnTF \c_novalue_tl { -NoValue- }`

is logically **false**. The `\c_novalue_tl` marker is intended for use in creating document-level interfaces, where it serves as an indicator that an (optional) argument was omitted. In particular, it is distinct from a simple empty `tl`.

<hr/> <code>\c_space_tl</code> <hr/>	An explicit space character contained in a token list (compare this with <code>\c_space_token</code>). For use where an explicit space is required.
--------------------------------------	--

13 Scratch token lists

<hr/> <code>\l_tmpa_tl</code> <code>\l_tmpb_tl</code> <hr/>	Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	--

<hr/> <code>\g_tmpa_tl</code> <code>\g_tmpb_tl</code> <hr/>	Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	---

Part VII

The l3str package: Strings

TeX associates each character with a category code: as such, there is no concept of a “string” as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense “ignoring” category codes: this is done by treating token lists as strings in a TeX sense.

A TeX string (and thus an expl3 string) is a series of characters which have category code 12 (“other”) with the exception of space characters which have category code 10 (“space”). Thus at a technical level, a TeX string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\tl_to_str:n` for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn’t primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) generate strings from the appropriate input: these are documented in `l3basics`, `l3tl` and `l3token`, respectively.

Most expandable functions in this module come in three flavours:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

1 Building strings

`\str_new:N``\str_new:c`

`New: 2015-09-18`

`\str_new:N <str var>`

Creates a new `<str var>` or raises an error if the name is already taken. The declaration is global. The `<str var>` is initially empty.

`\str_const:Nn``\str_const:(Nx|cn|cx)`

`New: 2015-09-18`

`\str_const:Nn <str var> {<token list>}`

Creates a new constant `<str var>` or raises an error if the name is already taken. The value of the `<str var>` is set globally to the `<token list>`, converted to a string.

```
\str_clear:N
\str_clear:c
\str_gclear:N
\str_gclear:c
```

New: 2015-09-18

`\str_clear:N` $\langle str\ var \rangle$
Clears the content of the $\langle str\ var \rangle$.

```
\str_clear_new:N
\str_clear_new:c
```

New: 2015-09-18

`\str_clear_new:N` $\langle str\ var \rangle$
Ensures that the $\langle str\ var \rangle$ exists globally by applying `\str_new:N` if necessary, then applies `\str_(g)clear:N` to leave the $\langle str\ var \rangle$ empty.

```
\str_set_eq:NN
\str_set_eq:(cN|Nc|cc)
\str_gset_eq:NN
\str_gset_eq:(cN|Nc|cc)
```

New: 2015-09-18

`\str_set_eq:NN` $\langle str\ var_1 \rangle$ $\langle str\ var_2 \rangle$
Sets the content of $\langle str\ var_1 \rangle$ equal to that of $\langle str\ var_2 \rangle$.

```
\str_concat:NNN
\str_concat:ccc
\str_gconcat:NNN
\str_gconcat:ccc
```

New: 2017-10-08

`\str_concat:NNN` $\langle str\ var_1 \rangle$ $\langle str\ var_2 \rangle$ $\langle str\ var_3 \rangle$
Concatenates the content of $\langle str\ var_2 \rangle$ and $\langle str\ var_3 \rangle$ together and saves the result in $\langle str\ var_1 \rangle$. The $\langle str\ var_2 \rangle$ is placed at the left side of the new string variable. The $\langle str\ var_2 \rangle$ and $\langle str\ var_3 \rangle$ must indeed be strings, as this function does not convert their contents to a string.

2 Adding data to string variables

```
\str_set:Nn
\str_set:(Nx|cn|cx)
\str_gset:Nn
\str_gset:(Nx|cn|cx)
```

New: 2015-09-18

`\str_set:Nn` $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$
Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and stores the result in $\langle str\ var \rangle$.

```
\str_put_left:Nn
\str_put_left:(Nx|cn|cx)
\str_gput_left:Nn
\str_gput_left:(Nx|cn|cx)
```

New: 2015-09-18

`\str_put_left:Nn` $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$
Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and prepends the result to $\langle str\ var \rangle$. The current contents of the $\langle str\ var \rangle$ are not automatically converted to a string.

```
\str_put_right:Nn
\str_put_right:(Nx|cn|cx)
\str_gput_right:Nn
\str_gput_right:(Nx|cn|cx)
```

New: 2015-09-18

`\str_put_right:Nn` $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$
Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and appends the result to $\langle str\ var \rangle$. The current contents of the $\langle str\ var \rangle$ are not automatically converted to a string.

3 Modifying string variables

```
\str_replace_once:Nnn
\str_replace_once:cnn
\str_greplace_once:Nnn
\str_greplace_once:cnn
```

New: 2017-10-08

```
\str_replace_once:Nnn <str var> {<old>} {<new>}
```

Converts the $\langle old \rangle$ and $\langle new \rangle$ token lists to strings, then replaces the first (leftmost) occurrence of $\langle old string \rangle$ in the $\langle str var \rangle$ with $\langle new string \rangle$.

```
\str_replace_all:Nnn
\str_replace_all:cnn
\str_greplace_all:Nnn
\str_greplace_all:cnn
```

New: 2017-10-08

```
\str_replace_all:Nnn <str var> {<old>} {<new>}
```

Converts the $\langle old \rangle$ and $\langle new \rangle$ token lists to strings, then replaces all occurrences of $\langle old string \rangle$ in the $\langle str var \rangle$ with $\langle new string \rangle$. As this function operates from left to right, the pattern $\langle old string \rangle$ may remain after the replacement (see `\str_remove_all:Nn` for an example).

```
\str_remove_once:Nn
\str_remove_once:cn
\str_gremove_once:Nn
\str_gremove_once:cn
```

New: 2017-10-08

```
\str_remove_once:Nn <str var> {<token list>}
```

Converts the $\langle token list \rangle$ to a $\langle string \rangle$ then removes the first (leftmost) occurrence of $\langle string \rangle$ from the $\langle str var \rangle$.

```
\str_remove_all:Nn
\str_remove_all:cn
\str_gremove_all:Nn
\str_gremove_all:cn
```

New: 2017-10-08

```
\str_remove_all:Nn <str var> {<token list>}
```

Converts the $\langle token list \rangle$ to a $\langle string \rangle$ then removes all occurrences of $\langle string \rangle$ from the $\langle str var \rangle$. As this function operates from left to right, the pattern $\langle string \rangle$ may remain after the removal, for instance,

```
\str_set:Nn \l_tmpa_str {abbccd} \str_remove_all:Nn \l_tmpa_str
{bc}
```

results in `\l_tmpa_str` containing `abcd`.

4 String conditionals

```
\str_if_exist_p:N ★
\str_if_exist_p:c ★
\str_if_exist:NTF ★
\str_if_exist:cTF ★
```

New: 2015-09-18

```
\str_if_exist_p:N <str var>
```

```
\str_if_exist:NTF <str var> {<true code>} {<false code>}
```

Tests whether the $\langle str var \rangle$ is currently defined. This does not check that the $\langle str var \rangle$ really is a string.

```
\str_if_empty_p:N ★
\str_if_empty_p:c ★
\str_if_empty:NTF ★
\str_if_empty:cTF ★
```

New: 2015-09-18

```
\str_if_empty_p:N <str var>
```

```
\str_if_empty:NTF <str var> {<true code>} {<false code>}
```

Tests if the $\langle string variable \rangle$ is entirely empty (*i.e.* contains no characters at all).

<code>\str_if_eq_p:NN</code>	★	<code>\str_if_eq_p:NN <str var₁> <str var₂></code>
<code>\str_if_eq_p:(Nc cN cc)</code>	★	<code>\str_if_eq:NNTF <str var₁> <str var₂> {<true code>} {<false code>}</code>
<code>\str_if_eq:NNTF</code>	★	
<code>\str_if_eq:(Nc cN cc)TF</code>	★	Compares the content of two <i><str variables></i> and is logically true if the two contain the same characters in the same order.

New: 2015-09-18

<code>\str_if_eq_p:nn</code>	★	<code>\str_if_eq_p:nn {<tl₁>} {<tl₂>}</code>
<code>\str_if_eq_p:(Vn on no nV VV)</code>	★	<code>\str_if_eq:nnTF {<tl₁>} {<tl₂>} {<true code>} {<false code>}</code>
<code>\str_if_eq:nnTF</code>	★	
<code>\str_if_eq:(Vn on no nV VV)TF</code>	★	

Compares the two *<token lists>* on a character by character basis (namely after converting them to strings), and is **true** if the two *<strings>* contain the same characters in the same order. Thus for example

`\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }`

is logically **true**.

<code>\str_if_eq_x_p:nn</code>	★	<code>\str_if_eq_x_p:nn {<tl₁>} {<tl₂>}</code>
<code>\str_if_eq_x_p:nnTF</code>	★	<code>\str_if_eq_x:nnTF {<tl₁>} {<tl₂>} {<true code>} {<false code>}</code>

New: 2012-06-05

Fully expands the two *<token lists>* and converts them to *<strings>*, then compares these on a character by character basis: it is **true** if the two *<strings>* contain the same characters in the same order. Thus for example

`\str_if_eq_x_p:nn { abc } { \tl_to_str:n { abc } }`

is logically **true**.

<code>\str_if_in:NnTF</code>		<code>\str_if_in:NnTF <str var> {<token list>} {<true code>} {<false code>}</code>
<code>\str_if_in:cnTF</code>		Converts the <i><token list></i> to a <i><string></i> and tests if that <i><string></i> is found in the content of the <i><str var></i> .

New: 2017-10-08

<code>\str_if_in:nnTF</code>		<code>\str_if_in:nnTF <tl₁> {<tl₂>} {<true code>} {<false code>}</code>
		Converts both <i><token lists></i> to <i><strings></i> and tests whether <i><string₂></i> is found inside <i><string₁></i> .

New: 2017-10-08

<code>\str_case:nn</code> ★	<code>\str_case:nnTF {⟨test string⟩}</code>
<code>\str_case:(on nV nv)</code> ★	{
<code>\str_case:nnTF</code> ★	{⟨string case ₁ ⟩} {⟨code case ₁ ⟩}
<code>\str_case:(on nV nv)TF</code> ★	{⟨string case ₂ ⟩} {⟨code case ₂ ⟩}
	...
	{⟨string case _n ⟩} {⟨code case _n ⟩}
	}
	{⟨true code⟩}
	{⟨false code⟩}

New: 2013-07-24
Updated: 2015-02-28

Compares the *⟨test string⟩* in turn with each of the *⟨string cases⟩* (all token lists are converted to strings). If the two are equal (as described for `\str_if_eq:nnTF`) then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

<code>\str_case_x:nn</code> ★	<code>\str_case_x:nnTF {⟨test string⟩}</code>
<code>\str_case_x:nnTF</code> ★	{
	{⟨string case ₁ ⟩} {⟨code case ₁ ⟩}
	{⟨string case ₂ ⟩} {⟨code case ₂ ⟩}
	...
	{⟨string case _n ⟩} {⟨code case _n ⟩}
	}
	{⟨true code⟩}
	{⟨false code⟩}

New: 2013-07-24

Compares the full expansion of the *⟨test string⟩* in turn with the full expansion of the *⟨string cases⟩* (all token lists are converted to strings). If the two full expansions are equal (as described for `\str_if_eq:nnTF`) then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\str_case_x:nn`, which does nothing if there is no match, is also available. The *⟨test string⟩* is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

5 Mapping to strings

<code>\str_map_function:NN</code> ☆	<code>\str_map_function:NN {⟨str var⟩} {⟨function⟩}</code>
<code>\str_map_function:cN</code> ☆	Applies <i>⟨function⟩</i> to every <i>⟨character⟩</i> in the <i>⟨str var⟩</i> including spaces. See also <code>\str_map_function:nN</code> .
<code>\str_map_function:nN</code> ☆	<code>\str_map_function:nN {⟨token list⟩} {⟨function⟩}</code>
	Converts the <i>⟨token list⟩</i> to a <i>⟨string⟩</i> then applies <i>⟨function⟩</i> to every <i>⟨character⟩</i> in the <i>⟨string⟩</i> including spaces. See also <code>\str_map_function:NN</code> .

New: 2017-11-14

New: 2017-11-14

<hr/> <code>\str_map_inline:Nn</code> <code>\str_map_inline:cn</code> <hr/> New: 2017-11-14	<code>\str_map_inline:Nn <str var> {<inline function>}</code> Applies the <i><inline function></i> to every <i><character></i> in the <i><str var></i> including spaces. The <i><inline function></i> should consist of code which receives the <i><character></i> as #1. See also <code>\str_map_function:NN</code> .
<hr/> <code>\str_map_inline:nn</code> <hr/> New: 2017-11-14	<code>\str_map_inline:nn {<token list>} {<inline function>}</code> Converts the <i><token list></i> to a <i><string></i> then applies the <i><inline function></i> to every <i><character></i> in the <i><string></i> including spaces. The <i><inline function></i> should consist of code which receives the <i><character></i> as #1. See also <code>\str_map_function:NN</code> .
<hr/> <code>\str_map_variable:NNn</code> <code>\str_map_variable:cNn</code> <hr/> New: 2017-11-14	<code>\str_map_variable:NNn <str var> <variable> {<code>}</code> Stores each <i><character></i> of the <i><string></i> (including spaces) in turn in the (string or token list) <i><variable></i> and applies the <i><code></i> . The <i><code></i> will usually make use of the <i><variable></i> , but this is not enforced. The assignments to the <i><variable></i> are local. See also <code>\str_map_inline:Nn</code> .
<hr/> <code>\str_map_variable:nNn</code> <hr/> New: 2017-11-14	<code>\str_map_variable:nNn {<token list>} <variable> {<code>}</code> Converts the <i><token list></i> to a <i><string></i> then stores each <i><character></i> in the <i><string></i> (including spaces) in turn in the (string or token list) <i><variable></i> and applies the <i><code></i> . The <i><code></i> will usually make use of the <i><variable></i> , but this is not enforced. The assignments to the <i><variable></i> are local. See also <code>\str_map_inline:Nn</code> .
<hr/> <code>\str_map_break: ☆</code> <hr/> New: 2017-10-08	<code>\str_map_break:</code> Used to terminate a <code>\str_map...</code> function before all characters in the <i><string></i> have been processed. This normally takes place within a conditional statement, for example <pre> \str_map_inline:Nn \l_my_str { \str_if_eq:nnT { #1 } { bingo } { \str_map_break: } % Do something useful } </pre> <p>See also <code>\str_map_break:n</code>. Use outside of a <code>\str_map...</code> scenario leads to low level \TeX errors.</p> <p>\TeXhackers note: When the mapping is broken, additional tokens may be inserted before continuing with the code that follows the loop. This depends on the design of the mapping function.</p>

`\str_map_break:n` ☆

New: 2017-10-08

`\str_map_break:n {⟨code⟩}`

Used to terminate a `\str_map...` function before all characters in the *⟨string⟩* have been processed, inserting the *⟨code⟩* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
{
  \str_if_eq:nnT { #1 } { bingo }
  { \str_map_break:n { <code> } }
  % Do something useful
}
```

Use outside of a `\str_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the *⟨code⟩* is inserted into the input stream. This depends on the design of the mapping function.

6 Working with the content of strings

`\str_use:N` ☆

`\str_use:c` ☆

New: 2015-09-18

`\str_use:N ⟨str var⟩`

Recovers the content of a *⟨str var⟩* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a *⟨str⟩* directly without an accessor function.

`\str_count:N`

`\str_count:c`

`\str_count:n`

`\str_count_ignore_spaces:n` ☆

New: 2015-09-18

☆ `\str_count:n {⟨token list⟩}`

Leaves in the input stream the number of characters in the string representation of *⟨token list⟩*, as an integer denotation. The functions differ in their treatment of spaces. In the case of `\str_count:N` and `\str_count:n`, all characters including spaces are counted. The `\str_count_ignore_spaces:n` function leaves the number of non-space characters in the input stream.

`\str_count_spaces:N` ☆

`\str_count_spaces:c` ☆

`\str_count_spaces:n` ☆

New: 2015-09-18

`\str_count_spaces:n {⟨token list⟩}`

Leaves in the input stream the number of space characters in the string representation of *⟨token list⟩*, as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

<code>\str_head:N</code>	★	<code>\str_head:n {⟨token list⟩}</code>
<code>\str_head:c</code>	★	
<code>\str_head:n</code>	★	
<code>\str_head_ignore_spaces:n</code>	★	

New: 2015-09-18

Converts the $\langle token list \rangle$ into a $\langle string \rangle$. The first character in the $\langle string \rangle$ is then left in the input stream, with category code “other”. The functions differ if the first character is a space: `\str_head:N` and `\str_head:n` return a space token with category code 10 (blank space), while the `\str_head_ignore_spaces:n` function ignores this space character and leaves the first non-space character in the input stream. If the $\langle string \rangle$ is empty (or only contains spaces in the case of the `_ignore_spaces` function), then nothing is left on the input stream.

<code>\str_tail:N</code>	★	<code>\str_tail:n {⟨token list⟩}</code>
<code>\str_tail:c</code>	★	
<code>\str_tail:n</code>	★	
<code>\str_tail_ignore_spaces:n</code>	★	

New: 2015-09-18

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: `\str_tail:N` and `\str_tail:n` only trim that space, while `\str_tail_ignore_spaces:n` removes the first non-space character and any space before it. If the $\langle token list \rangle$ is empty (or blank in the case of the `_ignore_spaces` variant), then nothing is left on the input stream.

<code>\str_item:Nn</code>	★	<code>\str_item:nn {⟨token list⟩} {⟨integer expression⟩}</code>
<code>\str_item:nn</code>	★	
<code>\str_item_ignore_spaces:nn</code>	★	

New: 2015-09-18

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the character in position $\langle integer expression \rangle$ of the $\langle string \rangle$, starting at 1 for the first (left-most) character. In the case of `\str_item:Nn` and `\str_item:nn`, all characters including spaces are taken into account. The `\str_item_ignore_spaces:nn` function skips spaces when counting characters. If the $\langle integer expression \rangle$ is negative, characters are counted from the end of the $\langle string \rangle$. Hence, -1 is the right-most character, *etc.*

```

\str_range:Nnn          ★ \str_range:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}
\str_range:cnn          ★
\str_range:nnn          ★
\str_range_ignore_spaces:nnn ★

```

New: 2015-09-18

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the characters from the $\langle start\ index \rangle$ to the $\langle end\ index \rangle$ inclusive. Positive $\langle indices \rangle$ are counted from the start of the string, 1 being the first character, and negative $\langle indices \rangle$ are counted from the end of the string, -1 being the last character. If either of $\langle start\ index \rangle$ or $\langle end\ index \rangle$ is 0, the result is empty. For instance,

```

\iow_term:x { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { 0 } { -1 } }

```

prints bcde, cdef, ef, and an empty line to the terminal. The $\langle start\ index \rangle$ must always be smaller than or equal to the $\langle end\ index \rangle$: if this is not the case then no output is generated. Thus

```

\iow_term:x { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:x { \str_range:nnn { abcdef } { -1 } { -4 } }

```

both yield empty strings.

7 String manipulation

<code>\str_lower_case:n</code>	★	<code>\str_lower_case:n {⟨tokens⟩}</code>
<code>\str_lower_case:f</code>	★	<code>\str_upper_case:n {⟨tokens⟩}</code>
<code>\str_upper_case:n</code>	★	
<code>\str_upper_case:f</code>	★	

New: 2015-03-01

Converts the input `⟨tokens⟩` to their string representation, as described for `\tl_to_str:n`, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file `UnicodeData.txt`.

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```
\cs_new_protected:Npn \myfunc:nn #1#2
{
  \cs_set_protected:cpn
  {
    user
    \str_upper_case:f { \tl_head:n {#1} }
    \str_lower_case:f { \tl_tail:n {#1} }
  }
  { #2 }
}
```

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should *not* be used for

- Caseless comparisons: use `\str_fold_case:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\tl_lower_case:n(n)`, `\tl_upper_case:n(n)` and `\tl_mixed_case:n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

T_EXhackers note: As with all `expl3` functions, the input supported by `\str_fold_case:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfT_EX *only* the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both X_ƎT_EX and LuaT_EX, subject only to the fact that X_ƎT_EX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

`\str_fold_case:n` ★ `\str_fold_case:n {(tokens)}`

`\str_fold_case:V` ★

New: 2014-06-19

Updated: 2016-03-07

Converts the input $\langle tokens \rangle$ to their string representation, as described for `\tl_to_str:n`, and then folds the case of the resulting $\langle string \rangle$ to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by `\str_fold_case:n` follows the mappings provided by the [Unicode Consortium](#), who [state](#):

Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by `\str_fold_case:n` follows the “full” scheme defined by the Unicode Consortium (*e.g.* SSfolds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (*i.e.* I always folds to i and not to ı).

TeXhackers note: As with all `expl3` functions, the input supported by `\str_fold_case:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTeX *only* the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both XeTeX and LuaTeX, subject only to the fact that XeTeX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

8 Viewing strings

`\str_show:N` `\str_show:N <str var>`

`\str_show:c`

`\str_show:n`

New: 2015-09-18

Displays the content of the $\langle str var \rangle$ on the terminal.

9 Constant token lists

`\c_ampersand_str`
`\c_atsign_str`
`\c_backslash_str`
`\c_left_brace_str`
`\c_right_brace_str`
`\c_circumflex_str`
`\c_colon_str`
`\c_dollar_str`
`\c_hash_str`
`\c_percent_str`
`\c_tilde_str`
`\c_underscore_str`

Constant strings, containing a single character token, with category code 12.

New: 2015-09-19

10 Scratch strings

`\l_tmpa_str`
`\l_tmpb_str`

Scratch strings for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_str`
`\g_tmpb_str`

Scratch strings for global assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Part VIII

The l3quark package

Quarks

Two special types of constants in L^AT_EX3 are “quarks” and “scan marks”. By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`.

1 Quarks

Quarks are control sequences that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, the most common use case being the ‘stop token’ (*i.e.* `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
{ <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster. An example of the quark testing functions and their use in recursion can be seen in the implementation of `\clist_map_function:NN`.

2 Defining quarks

`\quark_new:N`

`\quark_new:N <quark>`

Creates a new `<quark>` which expands only to `<quark>`. The `<quark>` is defined globally, and an error message is raised if the name was already taken.

`\q_stop`

Used as a marker for delimited arguments, such as

```
\cs_set:Npn \tmp:w #1#2 \q_stop {#1}
```

<u><u>\q_mark</u></u>	Used as a marker for delimited arguments when <code>\q_stop</code> is already in use.
<u><u>\q_nil</u></u>	Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself need to be tested (in contrast to <code>\q_stop</code> , which is only ever used as a delimiter).
<u><u>\q_no_value</u></u>	A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a “return” value by functions such as <code>\prop_get:NnN</code> if there is no data to return.

3 Quark tests

The method used to define quarks means that the single token (`N`) tests are faster than the multi-token (`n`) tests. The latter should therefore only be used when the argument can definitely take more than a single token.

<u><u>\quark_if_nil_p:N</u></u> *	<code>\quark_if_nil_p:N <token></code>
<u><u>\quark_if_nil:NTF</u></u> *	<code>\quark_if_nil:NTF <token> {\true code} {\false code}</code>

Tests if the `<token>` is equal to `\q_nil`.

<u><u>\quark_if_nil_p:n</u></u> *	<code>\quark_if_nil_p:n {\token list}</code>
<u><u>\quark_if_nil_p:(o V)</u></u> *	<code>\quark_if_nil:nTF {\token list} {\true code} {\false code}</code>
<u><u>\quark_if_nil:nTF</u></u> *	Tests if the <code><token list></code> contains only <code>\q_nil</code> (distinct from <code><token list></code> being empty or containing <code>\q_nil</code> plus one or more other tokens).
<u><u>\quark_if_nil:(o V)TF</u></u> *	

<u><u>\quark_if_no_value_p:N</u></u> *	<code>\quark_if_no_value_p:N <token></code>
<u><u>\quark_if_no_value_p:c</u></u> *	<code>\quark_if_no_value:NTF <token> {\true code} {\false code}</code>
<u><u>\quark_if_no_value:NTF</u></u> *	Tests if the <code><token></code> is equal to <code>\q_no_value</code> .
<u><u>\quark_if_no_value:cTF</u></u> *	

<u><u>\quark_if_no_value_p:n</u></u> *	<code>\quark_if_no_value_p:n {\token list}</code>
<u><u>\quark_if_no_value:nTF</u></u> *	<code>\quark_if_no_value:nTF {\token list} {\true code} {\false code}</code>

Tests if the `<token list>` contains only `\q_no_value` (distinct from `<token list>` being empty or containing `\q_no_value` plus one or more other tokens).

4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 5.

<u><u>\q_recursion_tail</u></u>	This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.
<u><u>\q_recursion_stop</u></u>	This quark is added <i>after</i> the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

`\quark_if_recursion_tail_stop:N` `\quark_if_recursion_tail_stop:N <token>`

Tests if *<token>* contains only the marker `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop:n` `\quark_if_recursion_tail_stop:n {<token list>}`
`\quark_if_recursion_tail_stop:o`

Updated: 2011-09-06

Tests if the *<token list>* contains only `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop_do:Nn` `\quark_if_recursion_tail_stop_do:Nn <token> {<insertion>}`

Tests if *<token>* contains only the marker `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The *<insertion>* code is then added to the input stream after the recursion has ended.

`\quark_if_recursion_tail_stop_do:nn` `\quark_if_recursion_tail_stop_do:nn {<token list>} {<insertion>}`
`\quark_if_recursion_tail_stop_do:on`

Updated: 2011-09-06

Tests if the *<token list>* contains only `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The *<insertion>* code is then added to the input stream after the recursion has ended.

`\quark_if_recursion_tail_break:NN` `\quark_if_recursion_tail_break:nn {<token list>} \<type>_map_break:`
`\quark_if_recursion_tail_break:nN`

New: 2018-04-10

Tests if *<token list>* contains only `\q_recursion_tail`, and if so terminates the recursion using `\<type>_map_break:.` The recursion end should be marked by `\prg_break_point:Nn \<type>_map_break:.`

5 An example of recursion with quarks

Quarks are mainly used internally in the `expl3` code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to use quarks in this fashion. We shall define a command called `\my_map_dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_dbl:nn {abcd} {[--#1--#2--]~}` would produce “`[-a-b-] [-c-d-]`”. Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here's the definition of `\my_map_dbl:nn`. First of all, define the function that does the processing based on the inline function argument `#2`. Then initiate the recursion using an internal function. The token list `#1` is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```
\cs_new:Npn \my_map_dbl:nn #1#2
{
  \cs_set:Npn \__my_map_dbl_fn:nn ##1 ##2 {#2}
  \__my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail
  \q_recursion_stop
}
```

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

```
\cs_new:Nn \__my_map_dbl:nn
{
  \quark_if_recursion_tail_stop:n {#1}
  \quark_if_recursion_tail_stop:n {#2}
  \__my_map_dbl_fn:nn {#1} {#2}
}
```

Finally, recurse:

```
\__my_map_dbl:nn
}
```

Note that contrarily to L^AT_EX3 built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of `__my_map_dbl_fn:nn`.

6 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence never expand in an expansion context and are (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by T_EX in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see l3regex).

<hr/> <code>\scan_new:N</code> <hr/>	<code>\scan_new:N</code> $\langle scan\ mark \rangle$
<hr/> <small>New: 2018-04-01</small> <hr/>	Creates a new $\langle scan\ mark \rangle$ which is set equal to <code>\scan_stop:</code> . The $\langle scan\ mark \rangle$ is defined globally, and an error message is raised if the name was already taken by another scan mark.
<hr/> <code>\s_stop</code> <hr/>	Used at the end of a set of instructions, as a marker that can be jumped to using <code>\use_none_delimit_by_s_stop:w</code> .
<hr/> <small>New: 2018-04-01</small> <hr/>	

<code>\use_none_delimit_by_s_stop:w</code>	<code>\use_none_delimit_by_s_stop:w</code> $\langle tokens \rangle$ <code>\s_stop</code>
--	--

New: 2018-04-01

Removes the $\langle tokens \rangle$ and `\s_stop` from the input stream. This leads to a low-level T_EX error if `\s_stop` is absent.

Part IX

The l3seq package

Sequences and stacks

L^AT_EX3 implements a “sequence” data type, which contain an ordered list of entries which may contain any *balanced text*. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in L^AT_EX3. This is achieved using a number of dedicated stack functions.

1 Creating and initialising sequences

<code>\seq_new:N</code>	<code>\seq_new:N <sequence></code>
<code>\seq_new:c</code>	

Creates a new *<sequence>* or raises an error if the name is already taken. The declaration is global. The *<sequence>* initially contains no items.

<code>\seq_clear:N</code>	<code>\seq_clear:N <sequence></code>
<code>\seq_clear:c</code>	
<code>\seq_gclear:N</code>	Clears all items from the <i><sequence></i> .
<code>\seq_gclear:c</code>	

<code>\seq_clear_new:N</code>	<code>\seq_clear_new:N <sequence></code>
<code>\seq_clear_new:c</code>	
<code>\seq_gclear_new:N</code>	Ensures that the <i><sequence></i> exists globally by applying <code>\seq_new:N</code> if necessary, then applies <code>\seq_(g)clear:N</code> to leave the <i><sequence></i> empty.
<code>\seq_gclear_new:c</code>	

<code>\seq_set_eq:NN</code>	<code>\seq_set_eq:NN <sequence₁> <sequence₂></code>
<code>\seq_set_eq:(cN Nc cc)</code>	
<code>\seq_gset_eq:NN</code>	Sets the content of <i><sequence₁></i> equal to that of <i><sequence₂></i> .
<code>\seq_gset_eq:(cN Nc cc)</code>	

<code>\seq_set_from_clist:NN</code>	<code>\seq_set_from_clist:NN <sequence> <comma-list></code>
<code>\seq_set_from_clist:(cN Nc cc)</code>	
<code>\seq_set_from_clist:Nn</code>	
<code>\seq_set_from_clist:cn</code>	
<code>\seq_gset_from_clist:NN</code>	
<code>\seq_gset_from_clist:(cN Nc cc)</code>	
<code>\seq_gset_from_clist:Nn</code>	
<code>\seq_gset_from_clist:cn</code>	

New: 2014-07-17

Converts the data in the *<comma list>* into a *<sequence>*: the original *<comma list>* is unchanged.

```
\seq_set_split:Nnn
\seq_set_split:NnV
\seq_gset_split:Nnn
\seq_gset_split:NnV
```

New: 2011-08-15
Updated: 2012-07-02

```
\seq_set_split:Nnn <sequence> {<delimiter>} {<token list>}
```

Splits the $\langle token list \rangle$ into $\langle items \rangle$ separated by $\langle delimiter \rangle$, and assigns the result to the $\langle sequence \rangle$. Spaces on both sides of each $\langle item \rangle$ are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of `l3clist` functions. Empty $\langle items \rangle$ are preserved by `\seq_set_split:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn <sequence> {<>}`. The $\langle delimiter \rangle$ may not contain `{`, `}` or `#` (assuming \TeX 's normal category code régime). If the $\langle delimiter \rangle$ is empty, the $\langle token list \rangle$ is split into $\langle items \rangle$ as a $\langle token list \rangle$.

```
\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc
```

```
\seq_concat:NNN <sequence1> <sequence2> <sequence3>
```

Concatenates the content of $\langle sequence_2 \rangle$ and $\langle sequence_3 \rangle$ together and saves the result in $\langle sequence_1 \rangle$. The items in $\langle sequence_2 \rangle$ are placed at the left side of the new sequence.

```
\seq_if_exist_p:N *
\seq_if_exist_p:c *
\seq_if_exist:NTF *
\seq_if_exist:cTF *
```

New: 2012-03-03

```
\seq_if_exist_p:N <sequence>
```

```
\seq_if_exist:NNTF <sequence> {<true code>} {<false code>}
```

Tests whether the $\langle sequence \rangle$ is currently defined. This does not check that the $\langle sequence \rangle$ really is a sequence variable.

2 Appending data to sequences

```
\seq_put_left:Nn
\seq_put_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gput_left:Nn
\seq_gput_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
```

```
\seq_put_left:Nn <sequence> {<item>}
```

Appends the $\langle item \rangle$ to the left of the $\langle sequence \rangle$.

```
\seq_put_right:Nn
\seq_put_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gput_right:Nn
\seq_gput_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
```

```
\seq_put_right:Nn <sequence> {<item>}
```

Appends the $\langle item \rangle$ to the right of the $\langle sequence \rangle$.

3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, *i.e.* setting the $\langle token list variable \rangle$ used with `\tl_set:Nn` and *never* `\tl_gset:Nn`.

```
\seq_get_left:NN
\seq_get_left:cN
```

Updated: 2012-05-14

```
\seq_get_left:NN <sequence> <token list variable>
```

Stores the left-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token list variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker `\q_no_value`.

<hr/> <code>\seq_get_right:NN</code> <code>\seq_get_right:cN</code> <hr/> Updated: 2012-05-19	<code>\seq_get_right:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Stores the right-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token list variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_pop_left:NN</code> <code>\seq_pop_left:cN</code> <hr/> Updated: 2012-05-14	<code>\seq_pop_left:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_gpop_left:NN</code> <code>\seq_gpop_left:cN</code> <hr/> Updated: 2012-05-14	<code>\seq_gpop_left:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_pop_right:NN</code> <code>\seq_pop_right:cN</code> <hr/> Updated: 2012-05-19	<code>\seq_pop_right:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_gpop_right:NN</code> <code>\seq_gpop_right:cN</code> <hr/> Updated: 2012-05-19	<code>\seq_gpop_right:NN</code> $\langle sequence \rangle$ $\langle token list variable \rangle$ Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the sequence and stores it in the $\langle token list variable \rangle$. The $\langle sequence \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local. If $\langle sequence \rangle$ is empty the $\langle token list variable \rangle$ is set to the special marker <code>\q_no_value</code> .
<hr/> <code>\seq_item:Nn</code> ★ <code>\seq_item:cn</code> ★ <hr/> New: 2014-07-17	<code>\seq_item:Nn</code> $\langle sequence \rangle$ $\{\langle integer expression \rangle\}$ Indexing items in the $\langle sequence \rangle$ from 1 at the top (left), this function evaluates the $\langle integer expression \rangle$ and leaves the appropriate item from the sequence in the input stream. If the $\langle integer expression \rangle$ is negative, indexing occurs from the bottom (right) of the sequence. If the $\langle integer expression \rangle$ is larger than the number of items in the $\langle sequence \rangle$ (as calculated by <code>\seq_count:N</code>) then the function expands to nothing.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an x-type argument expansion.

4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

<hr/> <code>\seq_get_left:NNTF</code> <code>\seq_get_left:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19 <hr/>	<code>\seq_get_left:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, stores the left-most item from the <i><sequence></i> in the <i><token list variable></i> without removing it from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. The <i><token list variable></i> is assigned locally.</p>
<hr/> <code>\seq_get_right:NNTF</code> <code>\seq_get_right:cNTF</code> <hr/> New: 2012-05-19 <hr/>	<code>\seq_get_right:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, stores the right-most item from the <i><sequence></i> in the <i><token list variable></i> without removing it from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. The <i><token list variable></i> is assigned locally.</p>
<hr/> <code>\seq_pop_left:NNTF</code> <code>\seq_pop_left:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19 <hr/>	<code>\seq_pop_left:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the left-most item from the <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. Both the <i><sequence></i> and the <i><token list variable></i> are assigned locally.</p>
<hr/> <code>\seq_gpop_left:NNTF</code> <code>\seq_gpop_left:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19 <hr/>	<code>\seq_gpop_left:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the left-most item from the <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. The <i><sequence></i> is modified globally, while the <i><token list variable></i> is assigned locally.</p>
<hr/> <code>\seq_pop_right:NNTF</code> <code>\seq_pop_right:cNTF</code> <hr/> New: 2012-05-19 <hr/>	<code>\seq_pop_right:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the right-most item from the <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. Both the <i><sequence></i> and the <i><token list variable></i> are assigned locally.</p>
<hr/> <code>\seq_gpop_right:NNTF</code> <code>\seq_gpop_right:cNTF</code> <hr/> New: 2012-05-19 <hr/>	<code>\seq_gpop_right:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code> <p>If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, pops the right-most item from the <i><sequence></i> in the <i><token list variable></i>, <i>i.e.</i> removes the item from the <i><sequence></i>, then leaves the <i><true code></i> in the input stream. The <i><sequence></i> is modified globally, while the <i><token list variable></i> is assigned locally.</p>

5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

```
\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
```

`\seq_remove_duplicates:N` $\langle sequence \rangle$

Removes duplicate items from the $\langle sequence \rangle$, leaving the left most copy of each item in the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

T_EXhackers note: This function iterates through every item in the $\langle sequence \rangle$ and does a comparison with the $\langle items \rangle$ already checked. It is therefore relatively slow with large sequences.

```
\seq_remove_all:Nn
\seq_remove_all:cn
\seq_gremove_all:Nn
\seq_gremove_all:cn
```

`\seq_remove_all:Nn` $\langle sequence \rangle$ $\{\langle item \rangle\}$

Removes every occurrence of $\langle item \rangle$ from the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

```
\seq_reverse:N
\seq_reverse:c
\seq_greverse:N
\seq_greverse:c
```

`\seq_reverse:N` $\langle sequence \rangle$

Reverses the order of the items stored in the $\langle sequence \rangle$.

New: 2014-07-18

```
\seq_sort:Nn
\seq_sort:cn
\seq_gsort:Nn
\seq_gsort:cn
```

`\seq_sort:Nn` $\langle sequence \rangle$ $\{\langle comparison code \rangle\}$

Sorts the items in the $\langle sequence \rangle$ according to the $\langle comparison code \rangle$, and assigns the result to $\langle sequence \rangle$. The details of sorting comparison are described in Section 1.

New: 2017-02-06

6 Sequence conditionals

```
\seq_if_empty_p:N ★
\seq_if_empty_p:c ★
\seq_if_empty:NnTF ★
\seq_if_empty:cnTF ★
```

`\seq_if_empty_p:N` $\langle sequence \rangle$

`\seq_if_empty:NnTF` $\langle sequence \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle sequence \rangle$ is empty (containing no items).

```
\seq_if_in:NnTF
\seq_if_in:(NV|Nv|No|Nx|cn|cV|cv|co|cx)TF
```

`\seq_if_in:NnTF` $\langle sequence \rangle$ $\{\langle item \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle item \rangle$ is present in the $\langle sequence \rangle$.

7 Mapping to sequences

`\seq_map_function:Nn` ☆

`\seq_map_function:cN` ☆

Updated: 2012-06-29

`\seq_map_function:Nn` $\langle sequence \rangle$ $\langle function \rangle$

Applies $\langle function \rangle$ to every $\langle item \rangle$ stored in the $\langle sequence \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. The $\langle items \rangle$ are returned from left to right. The function `\seq_map_inline:Nn` is faster than `\seq_map_function:Nn` for sequences with more than about 10 items.

`\seq_map_inline:Nn`

`\seq_map_inline:cN`

Updated: 2012-06-29

`\seq_map_inline:Nn` $\langle sequence \rangle$ $\{\langle inline function \rangle\}$

Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. The $\langle items \rangle$ are returned from left to right.

`\seq_map_variable:NNn`

`\seq_map_variable:(Ncn|cNn|ccn)`

Updated: 2012-06-29

`\seq_map_variable:NNn` $\langle sequence \rangle$ $\langle variable \rangle$ $\{\langle code \rangle\}$

Stores each $\langle item \rangle$ of the $\langle sequence \rangle$ in turn in the (token list) $\langle variable \rangle$ and applies the $\langle code \rangle$. The $\langle code \rangle$ will usually make use of the $\langle variable \rangle$, but this is not enforced. The assignments to the $\langle variable \rangle$ are local. The $\langle items \rangle$ are returned from left to right.

`\seq_map_break:` ☆

Updated: 2012-06-29

`\seq_map_break:`

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ have been processed. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \seq_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\seq_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\seq_map_break:n` ☆

Updated: 2012-06-29

`\seq_map_break:n {<code>}`

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ have been processed, inserting the $\langle code \rangle$ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \seq_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\seq_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the $\langle code \rangle$ is inserted into the input stream. This depends on the design of the mapping function.

`\seq_count:N` ☆

`\seq_count:c` ☆

New: 2012-07-13

`\seq_count:N <sequence>`

Leaves the number of items in the $\langle sequence \rangle$ in the input stream as an $\langle integer denotation \rangle$. The total number of items in a $\langle sequence \rangle$ includes those which are empty and duplicates, *i.e.* every item in a $\langle sequence \rangle$ is unique.

8 Using the content of sequences directly

`\seq_use:Nnnn` ☆

`\seq_use:cnnn` ☆

New: 2013-05-26

`\seq_use:Nnnn <seq var> {<separator between two>}`

`{<separator between more than two>} {<separator between final two>}`

Places the contents of the $\langle seq var \rangle$ in the input stream, with the appropriate $\langle separator \rangle$ between the items. Namely, if the sequence has more than two items, the $\langle separator between more than two \rangle$ is placed between each pair of items except the last, for which the $\langle separator between final two \rangle$ is used. If the sequence has exactly two items, then they are placed in the input stream separated by the $\langle separator between two \rangle$. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle items \rangle$ do not expand further when appearing in an x-type argument expansion.

<code>\seq_use:Nn</code> ★	<code>\seq_use:Nn <seq var> {<separator>}</code>
<code>\seq_use:cn</code> ★	Places the contents of the <i><seq var></i> in the input stream, with the <i><separator></i> between the items. If the sequence has a single item, it is placed in the input stream with no <i><separator></i> , and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.
New: 2013-05-26	

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<items>* do not expand further when appearing in an *x*-type argument expansion.

9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

<code>\seq_get:NN</code>	<code>\seq_get:NN <sequence> <token list variable></code>
<code>\seq_get:cn</code>	Reads the top item from a <i><sequence></i> into the <i><token list variable></i> without removing it from the <i><sequence></i> . The <i><token list variable></i> is assigned locally. If <i><sequence></i> is empty the <i><token list variable></i> is set to the special marker <code>\q_no_value</code> .
Updated: 2012-05-14	

<code>\seq_pop:NN</code>	<code>\seq_pop:NN <sequence> <token list variable></code>
<code>\seq_pop:cn</code>	Pops the top item from a <i><sequence></i> into the <i><token list variable></i> . Both of the variables are assigned locally. If <i><sequence></i> is empty the <i><token list variable></i> is set to the special marker <code>\q_no_value</code> .
Updated: 2012-05-14	

<code>\seq_gpop:NN</code>	<code>\seq_gpop:NN <sequence> <token list variable></code>
<code>\seq_gpop:cn</code>	Pops the top item from a <i><sequence></i> into the <i><token list variable></i> . The <i><sequence></i> is modified globally, while the <i><token list variable></i> is assigned locally. If <i><sequence></i> is empty the <i><token list variable></i> is set to the special marker <code>\q_no_value</code> .
Updated: 2012-05-14	

<code>\seq_get:NNTF</code>	<code>\seq_get:NNTF <sequence> <token list variable> {<true code>} {<false code>}</code>
<code>\seq_get:cNTF</code>	If the <i><sequence></i> is empty, leaves the <i><false code></i> in the input stream. The value of the <i><token list variable></i> is not defined in this case and should not be relied upon. If the <i><sequence></i> is non-empty, stores the top item from a <i><sequence></i> in the <i><token list variable></i> without removing it from the <i><sequence></i> . The <i><token list variable></i> is assigned locally.
New: 2012-05-14	
Updated: 2012-05-19	

`\seq_pop:NNTF`
`\seq_pop:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_pop:NNTF <sequence> <token list variable> {\true code} {\false code}`

If the `<sequence>` is empty, leaves the `<false code>` in the input stream. The value of the `<token list variable>` is not defined in this case and should not be relied upon. If the `<sequence>` is non-empty, pops the top item from the `<sequence>` in the `<token list variable>`, *i.e.* removes the item from the `<sequence>`. Both the `<sequence>` and the `<token list variable>` are assigned locally.

`\seq_gpop:NNTF`
`\seq_gpop:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_gpop:NNTF <sequence> <token list variable> {\true code} {\false code}`

If the `<sequence>` is empty, leaves the `<false code>` in the input stream. The value of the `<token list variable>` is not defined in this case and should not be relied upon. If the `<sequence>` is non-empty, pops the top item from the `<sequence>` in the `<token list variable>`, *i.e.* removes the item from the `<sequence>`. The `<sequence>` is modified globally, while the `<token list variable>` is assigned locally.

`\seq_push:Nn`
`\seq_push:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`
`\seq_gpush:Nn`
`\seq_gpush:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`

`\seq_push:Nn <sequence> {\item}`

Adds the `{\item}` to the top of the `<sequence>`.

10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurrences of a given item. To make sure that a `<sequence variable>` only has distinct items, use `\seq_remove_duplicates:N <sequence variable>`. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set `<seq var>` are straightforward. For instance, `\seq_count:N <seq var>` expands to the number of items, while `\seq_if_in:NnTF <seq var> {\item}` tests if the `<item>` is in the set.

Adding an `<item>` to a set `<seq var>` can be done by appending it to the `<seq var>` if it is not already in the `<seq var>`:

```
\seq_if_in:NnF <seq var> {\item}
{ \seq_put_right:Nn <seq var> {\item} }
```

Removing an `<item>` from a set `<seq var>` can be done using `\seq_remove_all:Nn`,

```
\seq_remove_all:Nn <seq var> {\item}
```

The intersection of two sets `<seq var1>` and `<seq var2>` can be stored into `<seq var3>` by collecting items of `<seq var1>` which are in `<seq var2>`.

```

\seq_clear:N <seq var3>
\seq_map_inline:Nn <seq var1>
{
\seq_if_in:NnT <seq var2> {#1}
{ \seq_put_right:Nn <seq var3> {#1} }
}

```

The code as written here only works if $\langle seq\ var_3 \rangle$ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence $\backslash l_ \langle pkg \rangle_internal_seq$, then $\langle seq\ var_3 \rangle$ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ through

```

\seq_concat:NNN <seq var3> <seq var1> <seq var2>
\seq_remove_duplicates:N <seq var3>

```

or by adding items to (a copy of) $\langle seq\ var_1 \rangle$ one by one

```

\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{
\seq_if_in:NnF <seq var3> {#1}
{ \seq_put_right:Nn <seq var3> {#1} }
}

```

The second approach is faster than the first when the $\langle seq\ var_2 \rangle$ is short compared to $\langle seq\ var_1 \rangle$.

The difference of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by removing items of the $\langle seq\ var_2 \rangle$ from (a copy of) the $\langle seq\ var_1 \rangle$ one by one.

```

\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn <seq var3> {#1} }

```

The symmetric difference of two sets $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ can be stored into $\langle seq\ var_3 \rangle$ by computing the difference between $\langle seq\ var_1 \rangle$ and $\langle seq\ var_2 \rangle$ and storing the result as $\backslash l_ \langle pkg \rangle_internal_seq$, then the difference between $\langle seq\ var_2 \rangle$ and $\langle seq\ var_1 \rangle$, and finally concatenating the two differences to get the symmetric differences.

```

\seq_set_eq:NN \l\_ \langle pkg \rangle\_internal\_seq <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn \l\_ \langle pkg \rangle\_internal\_seq {#1} }
\seq_set_eq:NN <seq var3> <seq var2>
\seq_map_inline:Nn <seq var1>
{ \seq_remove_all:Nn <seq var3> {#1} }
\seq_concat:NNN <seq var3> <seq var3> \l\_ \langle pkg \rangle\_internal\_seq

```

11 Constant and scratch sequences

$\backslash c_empty_seq$ Constant that is always empty.

New: 2012-07-02

`\l_tmpa_seq`
`\l_tmpb_seq`

New: 2012-04-26

Scratch sequences for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_seq`
`\g_tmpb_seq`

New: 2012-04-26

Scratch sequences for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

12 Viewing sequences

`\seq_show:N`
`\seq_show:c`

Updated: 2015-08-01

`\seq_show:N` $\langle sequence \rangle$
Displays the entries in the $\langle sequence \rangle$ in the terminal.

`\seq_log:N`
`\seq_log:c`

New: 2014-08-12
Updated: 2015-08-01

`\seq_log:N` $\langle sequence \rangle$
Writes the entries in the $\langle sequence \rangle$ in the log file.

Part X

The l3int package

Integers

Calculation and comparison of integer values can be carried out using literal numbers, `int` registers, constants and integers stored in token list variables. The standard operators `+`, `-`, `/` and `*` and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* (“`intexpr`”).

1 Integer expressions

`\int_eval:n` ★ `\int_eval:n {\langle integer expression \rangle}`

Evaluates the $\langle integer expression \rangle$, expanding any integer and token list variables within the $\langle expression \rangle$ to their content (without requiring `\int_use:N/\tl_use:N`) and applying the standard mathematical rules. For example both

`\int_eval:n { 5 + 4 * 3 - (3 + 4 * 5) }`

and

`\tl_new:N \l_my_tl`
`\tl_set:Nn \l_my_tl { 5 }`
`\int_new:N \l_my_int`
`\int_set:Nn \l_my_int { 4 }`
`\int_eval:n { \l_my_tl + \l_my_int * 3 - (3 + 4 * 5) }`

both evaluate to -6 . The $\{\langle integer expression \rangle\}$ may contain the operators `+`, `-`, `*` and `/`, along with parenthesis `(` and `)`. Any functions within the expressions should expand to an $\langle integer denotation \rangle$: a sequence of a sign and digits matching the regex `\-?[0-9]+`. After expansion `\int_eval:n` yields an $\langle integer denotation \rangle$ which is left in the input stream.

T_EXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an $\langle internal integer \rangle$, and therefore requires suitable termination if used in a T_EX-style integer assignment.

`\int_eval:w` ★ `\int_eval:w {\langle integer expression \rangle}`

New: 2018-03-30

Evaluates the $\langle integer expression \rangle$ as described for `\int_eval:n`. The end of the expression is the first token encountered that cannot form part of such an expression. If that token is `\scan_stop`: it is removed, otherwise not. Spaces do *not* terminate the expression.

`\int_abs:n` ★ `\int_abs:n {\langle integer expression \rangle}`

Updated: 2012-09-26

Evaluates the $\langle integer expression \rangle$ as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an $\langle integer denotation \rangle$ after two expansions.

<hr/>	
<code>\int_div_round:nn</code> ★	<code>\int_div_round:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-09-26	Evaluates the two $\langle integer expressions \rangle$ as described earlier, then divides the first value by the second, and rounds the result to the closest integer. Ties are rounded away from zero. Note that this is identical to using <code>/</code> directly in an $\langle integer expression \rangle$. The result is left in the input stream as an $\langle integer denotation \rangle$ after two expansions.
<hr/>	
<code>\int_div_truncate:nn</code> ★	<code>\int_div_truncate:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-02-09	Evaluates the two $\langle integer expressions \rangle$ as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using <code>/</code> rounds to the closest integer instead. The result is left in the input stream as an $\langle integer denotation \rangle$ after two expansions.
<hr/>	
<code>\int_max:nn</code> ★	<code>\int_max:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
<code>\int_min:nn</code> ★	<code>\int_min:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-09-26	Evaluates the $\langle integer expressions \rangle$ as described for <code>\int_eval:n</code> and leaves either the larger or smaller value in the input stream as an $\langle integer denotation \rangle$ after two expansions.
<hr/>	
<code>\int_mod:nn</code> ★	<code>\int_mod:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-09-26	Evaluates the two $\langle integer expressions \rangle$ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting <code>\int_div_truncate:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code> times $\langle integer \rangle_2$ from $\langle integer \rangle_1$. Thus, the result has the same sign as $\langle integer \rangle_1$ and its absolute value is strictly less than that of $\langle integer \rangle_2$. The result is left in the input stream as an $\langle integer denotation \rangle$ after two expansions.

2 Creating and initialising integers

<hr/>	
<code>\int_new:N</code>	<code>\int_new:N \langle integer \rangle</code>
<code>\int_new:c</code>	Creates a new $\langle integer \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle integer \rangle$ is initially equal to 0.
<hr/>	
<code>\int_const:Nn</code>	<code>\int_const:Nn \langle integer \rangle {\langle integer expression \rangle}</code>
<code>\int_const:cn</code>	Creates a new constant $\langle integer \rangle$ or raises an error if the name is already taken. The value of the $\langle integer \rangle$ is set globally to the $\langle integer expression \rangle$.
Updated: 2011-10-22	
<hr/>	
<code>\int_zero:N</code>	<code>\int_zero:N \langle integer \rangle</code>
<code>\int_zero:c</code>	Sets $\langle integer \rangle$ to 0.
<code>\int_gzero:N</code>	
<code>\int_gzero:c</code>	
<hr/>	
<code>\int_zero_new:N</code>	<code>\int_zero_new:N \langle integer \rangle</code>
<code>\int_zero_new:c</code>	Ensures that the $\langle integer \rangle$ exists globally by applying <code>\int_new:N</code> if necessary, then applies <code>\int_(g)zero:N</code> to leave the $\langle integer \rangle$ set to zero.
<code>\int_gzero_new:N</code>	
<code>\int_gzero_new:c</code>	
New: 2011-12-13	

```

\int_set_eq:NN
\int_set_eq:(cN|Nc|cc)
\int_gset_eq:NN
\int_gset_eq:(cN|Nc|cc)

```

`\int_set_eq:NN` $\langle integer_1 \rangle$ $\langle integer_2 \rangle$
Sets the content of $\langle integer_1 \rangle$ equal to that of $\langle integer_2 \rangle$.

```

\int_if_exist_p:N ★
\int_if_exist_p:c ★
\int_if_exist:NTF ★
\int_if_exist:cTF ★

```

`\int_if_exist_p:N` $\langle int \rangle$
`\int_if_exist:NTF` $\langle int \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
Tests whether the $\langle int \rangle$ is currently defined. This does not check that the $\langle int \rangle$ really is an integer variable.

New: 2012-03-03

3 Setting and incrementing integers

```

\int_add:Nn
\int_add:cn
\int_gadd:Nn
\int_gadd:cn

```

`\int_add:Nn` $\langle integer \rangle$ $\{\langle integer\ expression \rangle\}$
Adds the result of the $\langle integer\ expression \rangle$ to the current content of the $\langle integer \rangle$.

Updated: 2011-10-22

```

\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c

```

`\int_decr:N` $\langle integer \rangle$
Decreases the value stored in $\langle integer \rangle$ by 1.

```

\int_incr:N
\int_incr:c
\int_gincr:N
\int_gincr:c

```

`\int_incr:N` $\langle integer \rangle$
Increases the value stored in $\langle integer \rangle$ by 1.

```

\int_set:Nn
\int_set:cn
\int_gset:Nn
\int_gset:cn

```

`\int_set:Nn` $\langle integer \rangle$ $\{\langle integer\ expression \rangle\}$
Sets $\langle integer \rangle$ to the value of $\langle integer\ expression \rangle$, which must evaluate to an integer (as described for `\int_eval:n`).

Updated: 2011-10-22

```

\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

```

`\int_sub:Nn` $\langle integer \rangle$ $\{\langle integer\ expression \rangle\}$
Subtracts the result of the $\langle integer\ expression \rangle$ from the current content of the $\langle integer \rangle$.

Updated: 2011-10-22

4 Using integers

<code>\int_use:N</code>	★	<code>\int_use:N <integer></code>
<code>\int_use:c</code>	★	
<hr/>		
Updated: 2011-10-22		

T_EXhackers note: `\int_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

5 Integer expression conditionals

<code>\int_compare_p:nNn</code>	★	<code>\int_compare_p:nNn {<intexpr₁>} <relation> {<intexpr₂>}</code>
<code>\int_compare:nNnTF</code>	★	<code>\int_compare:nNnTF</code> <code>{<intexpr₁>} <relation> {<intexpr₂>}</code> <code>{<true code>} {<false code>}</code>

This function first evaluates each of the *<integer expressions>* as described for `\int_eval:n`. The two results are then compared using the *<relation>*:

Equal	=
Greater than	>
Less than	<

```

\int_compare_p:n ★ \int_compare_p:n
\int_compare:nTF ★ {
    <intexpr1> <relation1>
    ...
    <intexprN> <relationN>
    <intexprN+1>
}
\int_compare:nTF
{
    <intexpr1> <relation1>
    ...
    <intexprN> <relationN>
    <intexprN+1>
}
{<true code>} {<false code>}

```

Updated: 2013-01-13

This function evaluates the *<integer expressions>* as described for `\int_eval:n` and compares consecutive result using the corresponding *<relation>*, namely it compares *<intexpr₁>* and *<intexpr₂>* using the *<relation₁>*, then *<intexpr₂>* and *<intexpr₃>* using the *<relation₂>*, until finally comparing *<intexpr_N>* and *<intexpr_{N+1}>* using the *<relation_N>*. The test yields **true** if all comparisons are **true**. Each *<integer expression>* is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is **false**, then no other *<integer expression>* is evaluated and no other comparison is performed. The *<relations>* can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

<code>\int_case:nn</code> ★	<code>\int_case:nnTF {⟨test integer expression⟩}</code>
<code>\int_case:nnTF</code> ★	<code>{</code>
	<code>{⟨intexpr case₁⟩} {⟨code case₁⟩}</code>
	<code>{⟨intexpr case₂⟩} {⟨code case₂⟩}</code>
	<code>...</code>
	<code>{⟨intexpr case_n⟩} {⟨code case_n⟩}</code>
	<code>}</code>
	<code>{⟨true code⟩}</code>
	<code>{⟨false code⟩}</code>

New: 2013-07-24

This function evaluates the *⟨test integer expression⟩* and compares this in turn to each of the *⟨integer expression cases⟩*. If the two are equal then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\int_case:nn`, which does nothing if there is no match, is also available. For example

```
\int_case:nnF
{ 2 * 5 }
{
  { 5 }      { Small }
  { 4 + 6 }   { Medium }
  { -2 * 10 } { Negative }
}
{ No idea! }
```

leaves “Medium” in the input stream.

<code>\int_if_even_p:n</code> ★	<code>\int_if_odd_p:n {⟨integer expression⟩}</code>
<code>\int_if_even:nTF</code> ★	<code>\int_if_odd:nTF {⟨integer expression⟩}</code>
<code>\int_if_odd_p:n</code> ★	<code>{⟨true code⟩} {⟨false code⟩}</code>
<code>\int_if_odd:nTF</code> ★	

This function first evaluates the *⟨integer expression⟩* as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

6 Integer expression loops

<code>\int_do_until:nNnn</code> ☆	<code>\int_do_until:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨integer expressions⟩* as described for `\int_compare:nNnTF`. If the test is **false** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **true**.

<code>\int_do_while:nNnn</code> ☆	<code>\int_do_while:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨integer expressions⟩* as described for `\int_compare:nNnTF`. If the test is **true** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **false**.

<hr/> <code>\int_until_do:nNnn</code> ☆ <hr/>	<code>\int_until_do:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\int_while_do:nNnn</code> ☆ <hr/>	<code>\int_while_do:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .
<hr/> <code>\int_do_until:nn</code> ☆ <hr/>	<code>\int_do_until:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is false then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is true .
<hr/> <code>\int_do_while:nn</code> ☆ <hr/>	<code>\int_do_while:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is true then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is false .
<hr/> <code>\int_until_do:nn</code> ☆ <hr/>	<code>\int_until_do:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\int_while_do:nn</code> ☆ <hr/>	<code>\int_while_do:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .

7 Integer step functions

<code>\int_step_function:nN</code>	☆	<code>\int_step_function:nN {⟨final value⟩} ⟨function⟩</code>
<code>\int_step_function:nnN</code>	☆	<code>\int_step_function:nnN {⟨initial value⟩} {⟨final value⟩} ⟨function⟩</code>
<code>\int_step_function:nnnN</code>	☆	<code>\int_step_function:nnnN {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨function⟩</code>

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. The $\langle function \rangle$ is then placed in front of each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). The $\langle step \rangle$ must be non-zero. If the $\langle step \rangle$ is positive, the loop stops when the $\langle value \rangle$ becomes larger than the $\langle final\ value \rangle$. If the $\langle step \rangle$ is negative, the loop stops when the $\langle value \rangle$ becomes smaller than the $\langle final\ value \rangle$. The $\langle function \rangle$ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

```
[I saw 1]   [I saw 2]   [I saw 3]   [I saw 4]   [I saw 5]
```

The functions `\int_step_function:nN` and `\int_step_function:nnN` both use a fixed $\langle step \rangle$ of 1, and in the case of `\int_step_function:nN` the $\langle initial\ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

<code>\int_step_inline:nn</code>	<code>\int_step_inline:nn {⟨final value⟩} {⟨code⟩}</code>
<code>\int_step_inline:nnn</code>	<code>\int_step_inline:nnn {⟨initial value⟩} {⟨final value⟩} {⟨code⟩}</code>
<code>\int_step_inline:nnnn</code>	<code>\int_step_inline:nnnn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} {⟨code⟩}</code>

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream with `#1` replaced by the current $\langle value \rangle$. Thus the $\langle code \rangle$ should define a function of one argument (`#1`).

The functions `\int_step_inline:nn` and `\int_step_inline:nnn` both use a fixed $\langle step \rangle$ of 1, and in the case of `\int_step_inline:nn` the $\langle initial\ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

<code>\int_step_variable:nnn</code>	<code>\int_step_variable:nNn {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}</code>
<code>\int_step_variable:nnnn</code>	<code>\int_step_variable:nnNn {⟨initial value⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}</code>
<code>\int_step_variable:nnnnn</code>	<code>\int_step_variable:nnnNn {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨tl var⟩ {⟨code⟩}</code>

New: 2012-06-04
Updated: 2018-04-22

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. Then for each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream, with the $\langle tl\ var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl\ var \rangle$.

The functions `\int_step_variable:nNn` and `\int_step_variable:nnNn` both use a fixed $\langle step \rangle$ of 1, and in the case of `\int_step_variable:nNn` the $\langle initial\ value \rangle$ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

<code>\int_to_arabic:n</code> ★	<code>\int_to_arabic:n {⟨integer expression⟩}</code>
---------------------------------	--

Updated: 2011-10-22

Places the value of the $\langle integer\ expression \rangle$ in the input stream as digits, with category code 12 (other).

<code>\int_to_alph:n</code> ★	<code>\int_to_alph:n {⟨integer expression⟩}</code>
<code>\int_to_Alph:n</code> ★	

Updated: 2011-09-17

Evaluates the $\langle integer\ expression \rangle$ and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

```
\int_to_alph:n { 1 }
```

places a in the input stream,

```
\int_to_alph:n { 26 }
```

is represented as z and

```
\int_to_alph:n { 27 }
```

is converted to aa. For conversions using other alphabets, use `\int_to_symbols:nnn` to define an alphabet-specific function. The basic `\int_to_alph:n` and `\int_to_Alph:n` functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

<code>\int_to_symbols:nnn</code> ★	<code>\int_to_symbols:nnn</code> $\{ \langle integer\ expression \rangle \} \{ \langle total\ symbols \rangle \}$ $\{ \langle value\ to\ symbol\ mapping \rangle \}$
------------------------------------	--

Updated: 2011-09-17

This is the low-level function for conversion of an $\langle integer\ expression \rangle$ into a symbolic form (often letters). The $\langle total\ symbols \rangle$ available should be given as an integer expression. Values are actually converted to symbols according to the $\langle value\ to\ symbol\ mapping \rangle$. This should be given as $\langle total\ symbols \rangle$ pairs of entries, a number and the appropriate symbol. Thus the `\int_to_alph:n` function is defined as

```
\cs_new:Npn \int_to_alph:n #1
{
  \int_to_symbols:nnn {#1} { 26 }
  {
    { 1 } { a }
    { 2 } { b }
    ...
    { 26 } { z }
  }
}
```

<hr/>	
<code>\int_to_bin:n</code> ★	<code>\int_to_bin:n {⟨integer expression⟩}</code>
<hr/>	
New: 2014-02-11	Calculates the value of the <i>⟨integer expression⟩</i> and places the binary representation of the result in the input stream.
<hr/>	
<code>\int_to_hex:n</code> ★	<code>\int_to_hex:n {⟨integer expression⟩}</code>
<code>\int_to_Hex:n</code> ★	
<hr/>	
New: 2014-02-11	Calculates the value of the <i>⟨integer expression⟩</i> and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for <code>\int_to_hex:n</code> and upper case ones for <code>\int_to_Hex:n</code> . The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<hr/>	
<code>\int_to_oct:n</code> ★	<code>\int_to_oct:n {⟨integer expression⟩}</code>
<hr/>	
New: 2014-02-11	Calculates the value of the <i>⟨integer expression⟩</i> and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).
<hr/>	
<code>\int_to_base:nn</code> ★	<code>\int_to_base:nn {⟨integer expression⟩} {⟨base⟩}</code>
<code>\int_to_Base:nn</code> ★	
<hr/>	
Updated: 2014-02-11	Calculates the value of the <i>⟨integer expression⟩</i> and converts it into the appropriate representation in the <i>⟨base⟩</i> ; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by letters from the English alphabet: lower case letters for <code>\int_to_base:n</code> and upper case ones for <code>\int_to_Base:n</code> . The maximum <i>⟨base⟩</i> value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

TeXhackers note: This is a generic version of `\int_to_bin:n`, etc.

<hr/>	
<code>\int_to_roman:n</code> ☆	<code>\int_to_roman:n {⟨integer expression⟩}</code>
<code>\int_to_Roman:n</code> ☆	
<hr/>	
Updated: 2011-10-22	Places the value of the <i>⟨integer expression⟩</i> in the input stream as Roman numerals, either lower case (<code>\int_to_roman:n</code>) or upper case (<code>\int_to_Roman:n</code>). The Roman numerals are letters with category code 11 (letter).

9 Converting from other formats to integers

<hr/>	
<code>\int_from_alph:n</code> ★	<code>\int_from_alph:n {⟨letters⟩}</code>
<hr/>	
Updated: 2014-08-25	Converts the <i>⟨letters⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨letters⟩</i> are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with “a” equal to 1 through to “z” equal to 26. The function also accepts a leading sign, made of + and -. This is the inverse function of <code>\int_to_alph:n</code> and <code>\int_to_Alph:n</code> .
<hr/>	
<code>\int_from_bin:n</code> ★	<code>\int_from_bin:n {⟨binary number⟩}</code>
<hr/>	
New: 2014-02-11	Converts the <i>⟨binary number⟩</i> into the integer (base 10) representation and leaves this in the input stream. The <i>⟨binary number⟩</i> is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by binary digits. This is the inverse function of <code>\int_to_bin:n</code> .
Updated: 2014-08-25	

<hr/> <code>\int_from_hex:n</code> ★ <hr/>	<code>\int_from_hex:n</code> $\{\langle hexadecimal\ number\rangle\}$
New: 2014-02-11 Updated: 2014-08-25 <hr/>	Converts the $\langle hexadecimal\ number\rangle$ into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the $\langle hexadecimal\ number\rangle$ by upper or lower case letters. The $\langle hexadecimal\ number\rangle$ is first converted to a string, with no expansion. The function also accepts a leading sign, made of + and -. This is the inverse function of <code>\int_to_hex:n</code> and <code>\int_to_Hex:n</code> .

<hr/> <code>\int_from_oct:n</code> ★ <hr/>	<code>\int_from_oct:n</code> $\{\langle octal\ number\rangle\}$
New: 2014-02-11 Updated: 2014-08-25 <hr/>	Converts the $\langle octal\ number\rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle octal\ number\rangle$ is first converted to a string, with no expansion. The function accepts a leading sign, made of + and -, followed by octal digits. This is the inverse function of <code>\int_to_oct:n</code> .

<hr/> <code>\int_from_roman:n</code> ★ <hr/>	<code>\int_from_roman:n</code> $\{\langle roman\ numeral\rangle\}$
Updated: 2014-08-25 <hr/>	Converts the $\langle roman\ numeral\rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle roman\ numeral\rangle$ is first converted to a string, with no expansion. The $\langle roman\ numeral\rangle$ may be in upper or lower case; if the numeral contains characters besides mdclxvi or MDCLXVI then the resulting value is -1. This is the inverse function of <code>\int_to_roman:n</code> and <code>\int_to_Roman:n</code> .

<hr/> <code>\int_from_base:nn</code> ★ <hr/>	<code>\int_from_base:nn</code> $\{\langle number\rangle\}$ $\{\langle base\rangle\}$
Updated: 2014-08-25 <hr/>	Converts the $\langle number\rangle$ expressed in $\langle base\rangle$ into the appropriate value in base 10. The $\langle number\rangle$ is first converted to a string, with no expansion. The $\langle number\rangle$ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum $\langle base\rangle$ value is 36. This is the inverse function of <code>\int_to_base:nn</code> and <code>\int_to_Base:nn</code> .

10 Random integers

<hr/> <code>\int_rand:nn</code> ★ <hr/>	<code>\int_rand:nn</code> $\{\langle intexpr_1\rangle\}$ $\{\langle intexpr_2\rangle\}$
New: 2016-12-06 Updated: 2018-04-27 <hr/>	Evaluates the two $\langle integer\ expressions\rangle$ and produces a pseudo-random number between the two (with bounds included). This is not yet available in X _Y TeX.

11 Viewing integers

<hr/> <code>\int_show:N</code> <code>\int_show:c</code> <hr/>	<code>\int_show:N</code> $\langle integer\rangle$ Displays the value of the $\langle integer\rangle$ on the terminal.
--	--

<hr/> <code>\int_show:n</code> <hr/>	<code>\int_show:n</code> $\{\langle integer\ expression\rangle\}$
New: 2011-11-22 Updated: 2015-08-07 <hr/>	Displays the result of evaluating the $\langle integer\ expression\rangle$ on the terminal.

`\int_log:N`
`\int_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\int_log:N` $\langle integer \rangle$
Writes the value of the $\langle integer \rangle$ in the log file.

`\int_log:n`

New: 2014-08-22
Updated: 2015-08-07

`\int_log:n` $\{\langle integer expression \rangle\}$
Writes the result of evaluating the $\langle integer expression \rangle$ in the log file.

12 Constant integers

`\c_zero`
`\c_one`
`\c_two`
`\c_three`
`\c_four`
`\c_five`
`\c_six`
`\c_seven`
`\c_eight`
`\c_nine`
`\c_ten`
`\c_eleven`
`\c_twelve`
`\c_thirteen`
`\c_fourteen`
`\c_fifteen`
`\c_sixteen`
`\c_thirty_two`
`\c_one_hundred`
`\c_two_hundred_fifty_five`
`\c_two_hundred_fifty_six`
`\c_one_thousand`
`\c_ten_thousand`

Integer values used with primitive tests and assignments: self-terminating nature makes these more convenient and faster than literal numbers.

`\c_max_int`

The maximum value that can be stored as an integer.

`\c_max_register_int`

Maximum number of registers.

`\c_max_char_int`

Maximum character code completely supported by the engine.

13 Scratch integers

`\l_tmpa_int`
`\l_tmpb_int`

Scratch integer for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_int`
`\g_tmpb_int`

Scratch integer for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

13.1 Direct number expansion

`\int_value:w` ★
New: 2018-03-27

`\int_value:w` $\langle integer \rangle$
`\int_value:w` $\langle integer\ denotation \rangle$ $\langle optional\ space \rangle$

Expands the following tokens until an $\langle integer \rangle$ is formed, and leaves a normalized form (no leading sign except for negative numbers, no leading digit 0 except for zero) in the input stream as category code 12 (other) characters. The $\langle integer \rangle$ can consist of any number of signs (with intervening spaces) followed by

- an integer variable (in fact, any T_EX register except `\toks`) or
- explicit digits (or by ‘ $\langle octal\ digits \rangle$ ’ or “ $\langle hexadecimal\ digits \rangle$ ” or ‘ $\langle character \rangle$ ’).

In this last case expansion stops once a non-digit is found; if that is a space it is removed as in f-expansion, and so `\exp_stop_f:` may be employed as an end marker. Note that protected functions *are* expanded by this process.

This function requires exactly one expansion to produce a value, and so is suitable for use in cases where a number is required “directly”. In general, `\int_eval:n` is the preferred approach to generating numbers.

T_EXhackers note: This is the T_EX primitive `\number`.

14 Primitive conditionals

`\if_int_compare:w` ★

`\if_int_compare:w` $\langle integer_1 \rangle$ $\langle relation \rangle$ $\langle integer_2 \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle false\ code \rangle$
`\fi:`

Compare two integers using $\langle relation \rangle$, which must be one of =, < or > with category code 12. The `\else:` branch is optional.

T_EXhackers note: These are both names for the T_EX primitive `\ifnum`.

<code>\if_case:w</code>	★	<code>\if_case:w</code>	$\langle integer \rangle$	$\langle case_0 \rangle$
<code>\or:</code>	★		<code>\or:</code>	$\langle case_1 \rangle$
			<code>\or:</code>	<code>...</code>
			<code>\else:</code>	$\langle default \rangle$
		<code>\fi:</code>		

Selects a case to execute based on the value of the $\langle integer \rangle$. The first case ($\langle case_0 \rangle$) is executed if $\langle integer \rangle$ is 0, the second ($\langle case_1 \rangle$) if the $\langle integer \rangle$ is 1, *etc.* The $\langle integer \rangle$ may be a literal, a constant or an integer expression (*e.g.* using `\int_eval:n`).

T_EXhackers note: These are the T_EX primitives `\ifcase` and `\or`.

<code>\if_int_odd:w</code>	★	<code>\if_int_odd:w</code>	$\langle tokens \rangle$	$\langle optional\ space \rangle$
			$\langle true\ code \rangle$	
		<code>\else:</code>		
			$\langle true\ code \rangle$	
		<code>\fi:</code>		

Expands $\langle tokens \rangle$ until a non-numeric token or a space is found, and tests whether the resulting $\langle integer \rangle$ is odd. If so, $\langle true\ code \rangle$ is executed. The `\else:` branch is optional.

T_EXhackers note: This is the T_EX primitive `\ifodd`.

Part XI

The l3flag package: Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any non-negative value, which we call its *height*. In expansion-only contexts, a flag can only be “raised”: this increases the *height* by 1. The *height* can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local. They are referenced by a *flag name* such as `str_missing`. The *flag name* is used as part of `\use:c` constructions hence is expanded at point of use. It must expand to character tokens only, with no spaces.

A typical use case of flags would be to keep track of whether an exceptional condition has occurred during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by `l3str-convert`, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height. Flags should not be used unless unavoidable.

1 Setting up flags

<code>\flag_new:n</code>	<code>\flag_new:n {<flag name>}</code>
--------------------------	--

Creates a new flag with a name given by *flag name*, or raises an error if the name is already taken. The *flag name* may not contain spaces. The declaration is global, but flags are always local variables. The *flag* initially has zero height.

<code>\flag_clear:n</code>	<code>\flag_clear:n {<flag name>}</code>
----------------------------	--

The *flag*’s height is set to zero. The assignment is local.

<code>\flag_clear_new:n</code>	<code>\flag_clear_new:n {<flag name>}</code>
--------------------------------	--

Ensures that the *flag* exists globally by applying `\flag_new:n` if necessary, then applies `\flag_clear:n`, setting the height to zero locally.

<code>\flag_show:n</code>	<code>\flag_show:n {<flag name>}</code>
---------------------------	---

Displays the *flag*’s height in the terminal.

<code>\flag_log:n</code>	<code>\flag_log:n {<flag name>}</code>
--------------------------	--

Writes the *flag*’s height to the log file.

2 Expandable flag commands

<hr/> <code>\flag_if_exist:n</code> ★	<code>\flag_if_exist:n {⟨flag name⟩}</code>
<hr/> <code>\flag_if_exist:n</code> <u><code>TF</code></u> ★	This function returns <code>true</code> if the <code>⟨flag name⟩</code> references a flag that has been defined previously, and <code>false</code> otherwise.
<hr/> <code>\flag_if_raised:n</code> ★	<code>\flag_if_raised:n {⟨flag name⟩}</code>
<hr/> <code>\flag_if_raised:n</code> <u><code>TF</code></u> ★	This function returns <code>true</code> if the <code>⟨flag⟩</code> has non-zero height, and <code>false</code> if the <code>⟨flag⟩</code> has zero height.
<hr/> <code>\flag_height:n</code> ★	<code>\flag_height:n {⟨flag name⟩}</code>
	Expands to the height of the <code>⟨flag⟩</code> as an integer denotation.
<hr/> <code>\flag_raise:n</code> ★	<code>\flag_raise:n {⟨flag name⟩}</code>
	The <code>⟨flag⟩</code> 's height is increased by 1 locally.

Part XII

The l3prg package

Control structures

Conditional processing in L^AT_EX3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead in the input stream. After processing the input, a *state* is returned. The states returned are *⟨true⟩* and *⟨false⟩*.

L^AT_EX3 has two forms of conditional flow processing based on these states. The first form is predicate functions that turn the returned state into a boolean *⟨true⟩* or *⟨false⟩*. For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean *⟨true⟩* or *⟨false⟩* values to be used in testing with `\if_predicate:w` or in functions to be described below. The second form is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in `\cs_if_free:NTF` which also takes one argument (the *N*) and then executes either **true** or **false** depending on the result.

T_EXhackers note: The arguments are executed after exiting the underlying `\if... \fi` structure.

1 Defining a set of conditional functions

```
\prg_new_conditional:Npnn
\prg_set_conditional:Npnn
\prg_new_conditional:Nnn
\prg_set_conditional:Nnn
```

Updated: 2012-02-06

```
\prg_new_conditional:Npnn \<name>:<arg spec> <parameters> {\<conditions>} {\<code>}
\prg_new_conditional:Nnn \<name>:<arg spec> {\<conditions>} {\<code>}
```

These functions create a family of conditionals using the same *{⟨code⟩}* to perform the test created. Those conditionals are expandable if *⟨code⟩* is. The **new** versions check for existing definitions and perform assignments globally (cf. `\cs_new:Npn`) whereas the **set** versions do no check and perform assignments locally (cf. `\cs_set:Npn`). The conditionals created are dependent on the comma-separated list of *⟨conditions⟩*, which should be one or more of **p**, **T**, **F** and **TF**.

```
\prg_new_protected_conditional:Npnn \prg_new_protected_conditional:Npnn \<name>:<arg spec> <parameters>
\prg_set_protected_conditional:Npnn {\<conditions>} {\<code>}
\prg_new_protected_conditional:Nnn \prg_new_protected_conditional:Nnn \<name>:<arg spec>
\prg_set_protected_conditional:Nnn {\<conditions>} {\<code>}
```

Updated: 2012-02-06

These functions create a family of protected conditionals using the same *{⟨code⟩}* to perform the test created. The *⟨code⟩* does not need to be expandable. The **new** version check for existing definitions and perform assignments globally (cf. `\cs_new:Npn`) whereas the **set** version do not (cf. `\cs_set:Npn`). The conditionals created are depended on the comma-separated list of *⟨conditions⟩*, which should be one or more of **T**, **F** and **TF** (not **p**).

The conditionals are defined by `\prg_new_conditional:Npnn` and friends as:

- `\<name>_p:<arg spec>` — a predicate function which will supply either a logical `true` or logical `false`. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome. This function cannot be defined for `protected` conditionals.
- `\<name>:<arg spec>T` — a function with one more argument than the original `<arg spec>` demands. The `<true branch>` code in this additional argument will be left on the input stream only if the test is `true`.
- `\<name>:<arg spec>F` — a function with one more argument than the original `<arg spec>` demands. The `<false branch>` code in this additional argument will be left on the input stream only if the test is `false`.
- `\<name>:<arg spec>TF` — a function with two more argument than the original `<arg spec>` demands. The `<true branch>` code in the first additional argument will be left on the input stream if the test is `true`, while the `<false branch>` code in the second argument will be left on the input stream if the test is `false`.

The `<code>` of the test may use `<parameters>` as specified by the second argument to `\prg_set_conditional:Npnn`: this should match the `<argument specification>` but this is not enforced. The `Nnn` versions infer the number of arguments from the argument specification given (cf. `\cs_new:Nn`, etc.). Within the `<code>`, the functions `\prg_return_true:` and `\prg_return_false:` are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

```
\prg_set_conditional:Npnn \foo_if_bar:NN #1#2 { p , T , TF }
{
  \if_meaning:w \l_tmpa_tl #1
  \prg_return_true:
\else:
  \if_meaning:w \l_tmpa_tl #2
  \prg_return_true:
\else:
  \prg_return_false:
\fi:
\fi:
}
```

This defines the function `\foo_if_bar_p:NN`, `\foo_if_bar:NNTF` and `\foo_if_bar:NNT` but not `\foo_if_bar:NNF` (because `F` is missing from the `<conditions>` list). The return statements take care of resolving the remaining `\else:` and `\fi:` before returning the state. There must be a return statement for each branch; failing to do so will result in erroneous output if that branch is executed.

<code>\prg_new_eq_conditional:Nnn</code>	<code>\prg_new_eq_conditional:Nnn \<name1>:<arg spec1> \<name2>:<arg spec2></code>
<code>\prg_set_eq_conditional:Nnn</code>	<code>{<conditions>}</code>

These functions copy a family of conditionals. The `new` version checks for existing definitions (cf. `\cs_new_eq:NN`) whereas the `set` version does not (cf. `\cs_set_eq:NN`). The conditionals copied are depended on the comma-separated list of `<conditions>`, which should be one or more of `p`, `T`, `F` and `TF`.

<code>\prg_return_true:</code>	★	<code>\prg_return_true:</code>
<code>\prg_return_false:</code>	★	<code>\prg_return_false:</code>

These “return” functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npnn`, *etc.*, to indicate when a true or false branch should be taken. While they may appear multiple times each within the code of such conditionals, the execution of the conditional must result in the expansion of one of these two functions *exactly once*.

The return functions trigger what is internally an **f**-expansion process to complete the evaluation of the conditional. Therefore, after `\prg_return_true:` or `\prg_return_false:` there must be no non-expandable material in the input stream for the remainder of the expansion of the conditional code. This includes other instances of either of these functions.

2 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (*e.g.*, draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and `\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations And, Or, Not, *etc.* which can then be used on both the boolean type and predicate functions.

All conditional `\bool_` functions except assignments are expandable and expect the input to also be fully expandable (which generally means being constructed from predicate functions and booleans, possibly nested).

T_EXhackers note: The `bool` data type is not implemented using the `\iffalse/\iftrue` primitives, in contrast to `\newif`, *etc.*, in plain T_EX, L^AT_EX 2_ε and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

<code>\bool_new:N</code>	<code>\bool_new:N</code>	<code><boolean></code>
<code>\bool_new:c</code>		

Creates a new `<boolean>` or raises an error if the name is already taken. The declaration is global. The `<boolean>` is initially **false**.

<code>\bool_set_false:N</code>	<code>\bool_set_false:N</code>	<code><boolean></code>
<code>\bool_set_false:c</code>		
<code>\bool_gset_false:N</code>		
<code>\bool_gset_false:c</code>		

Sets `<boolean>` logically **false**.

<code>\bool_set_true:N</code>	<code>\bool_set_true:N</code>	<code><boolean></code>
<code>\bool_set_true:c</code>		
<code>\bool_gset_true:N</code>		
<code>\bool_gset_true:c</code>		

Sets `<boolean>` logically **true**.

```
\bool_set_eq:NN
\bool_set_eq:(cN|Nc|cc)
\bool_gset_eq:NN
\bool_gset_eq:(cN|Nc|cc)
```

`\bool_set_eq:NN` $\langle boolean_1 \rangle$ $\langle boolean_2 \rangle$
 Sets $\langle boolean_1 \rangle$ to the current value of $\langle boolean_2 \rangle$.

```
\bool_set:Nn
\bool_set:cn
\bool_gset:Nn
\bool_gset:cn
```

`\bool_set:Nn` $\langle boolean \rangle$ $\{\langle boolexpr \rangle\}$
`\bool_set:cn`
`\bool_gset:Nn`
`\bool_gset:cn`
 Evaluates the $\langle boolean \text{ expression} \rangle$ as described for `\bool_if:nTF`, and sets the $\langle boolean \rangle$ variable to the logical truth of this evaluation.

Updated: 2017-07-15

```
\bool_if_p:N *
\bool_if_p:c *
\bool_if:NTF *
\bool_if:cTF *
```

`\bool_if_p:N` $\langle boolean \rangle$
`\bool_if:NTF` $\langle boolean \rangle$ $\{\langle true \text{ code} \rangle\}$ $\{\langle false \text{ code} \rangle\}$
`\bool_if:NTF` *
`\bool_if:cTF` *
 Tests the current truth of $\langle boolean \rangle$, and continues expansion based on this result.

Updated: 2017-07-15

```
\bool_show:N
\bool_show:c
```

`\bool_show:N` $\langle boolean \rangle$
`\bool_show:c`
 Displays the logical truth of the $\langle boolean \rangle$ on the terminal.

New: 2012-02-09

Updated: 2015-08-01

```
\bool_show:n
```

`\bool_show:n` $\{\langle boolean \text{ expression} \rangle\}$

New: 2012-02-09

Updated: 2017-07-15

```
\bool_log:N
\bool_log:c
```

`\bool_log:N` $\langle boolean \rangle$
`\bool_log:c`
 Writes the logical truth of the $\langle boolean \rangle$ in the log file.

New: 2014-08-22

Updated: 2015-08-03

```
\bool_log:n
```

`\bool_log:n` $\{\langle boolean \text{ expression} \rangle\}$

New: 2014-08-22

Updated: 2017-07-15

```
\bool_if_exist_p:N *
\bool_if_exist_p:c *
\bool_if_exist:NTF *
\bool_if_exist:cTF *
```

`\bool_if_exist_p:N` $\langle boolean \rangle$
`\bool_if_exist:NTF` $\langle boolean \rangle$ $\{\langle true \text{ code} \rangle\}$ $\{\langle false \text{ code} \rangle\}$
`\bool_if_exist:NTF` *
`\bool_if_exist:cTF` *
 Tests whether the $\langle boolean \rangle$ is currently defined. This does not check that the $\langle boolean \rangle$ really is a boolean variable.

New: 2012-03-03

```
\l_tmpa_bool
\l_tmpb_bool
```

A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any L^AT_EX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_bool`
`\g_tmpb_bool`

A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any L^AT_EX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean $\langle true \rangle$ or $\langle false \rangle$ values, it seems only fitting that we also provide a parser for $\langle boolean\ expressions \rangle$.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean $\langle true \rangle$ or $\langle false \rangle$. It supports the logical operations And, Or and Not as the well-known infix operators `&&` and `||` and prefix `!` with their usual precedences (namely, `&&` binds more tightly than `||`). In addition to this, parentheses can be used to isolate sub-expressions. For example,

```
\int_compare_p:n { 1 = 1 } &&
(
  \int_compare_p:n { 2 = 3 } ||
  \int_compare_p:n { 4 <= 4 } ||
  \str_if_eq_p:nn { abc } { def }
) &&
! \int_compare_p:n { 2 = 4 }
```

is a valid boolean expression.

Contrarily to some other programming languages, the operators `&&` and `||` evaluate both operands in all cases, even when the first operand is enough to determine the result. This “eager” evaluation should be contrasted with the “lazy” evaluation of `\bool_lazy_...` functions.

T_EXhackers note: The eager evaluation of boolean expressions is unfortunately necessary in T_EX. Indeed, a lazy parser can get confused if `&&` or `||` or parentheses appear as (unbraced) arguments of some predicates. For instance, the innocuous-looking expression below would break (in a lazy parser) if `#1` were a closing parenthesis and `\l_tmpa_bool` were `true`.

```
( \l_tmpa_bool || \token_if_eq_meaning_p:NN X #1 )
```

Minimal (lazy) evaluation can be obtained using the conditionals `\bool_lazy_all:nTF`, `\bool_lazy_and:nnTF`, `\bool_lazy_any:nTF`, or `\bool_lazy_or:nnTF`, which only evaluate their boolean expression arguments when they are needed to determine the resulting truth value. For example, when evaluating the boolean expression

```
\bool_lazy_and_p:nn
{
  \bool_lazy_any_p:n
  {
    { \int_compare_p:n { 2 = 3 } }
    { \int_compare_p:n { 4 <= 4 } }
    { \int_compare_p:n { 1 = \error } } % skipped
  }
}
{ ! \int_compare_p:n { 2 = 4 } }
```


the line marked with `skipped` is not expanded because the result of `\bool_lazy_any_p:n` is known once the second boolean expression is found to be logically `true`. On the other hand, the last line is expanded because its logical value is needed to determine the result of `\bool_lazy_and_p:nn`.

<code>\bool_if_p:n</code> ★ <code>\bool_if:nTF</code> ★ <hr/> Updated: 2017-07-15	<code>\bool_if_p:n {⟨boolean expression⟩}</code> <code>\bool_if:nTF {⟨boolean expression⟩} {⟨true code⟩} {⟨false code⟩}</code>
---	---

Tests the current truth of *⟨boolean expression⟩*, and continues expansion based on this result. The *⟨boolean expression⟩* should consist of a series of predicates or boolean variables with the logical relationship between these defined using `&&` (“And”), `||` (“Or”), `!` (“Not”) and parentheses. The logical Not applies to the next predicate or group.

<code>\bool_lazy_all_p:n</code> ★ <code>\bool_lazy_all:nTF</code> ★ <hr/> New: 2015-11-15 Updated: 2017-07-15	<code>\bool_lazy_all_p:n { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ... {⟨boolexpr_N⟩} }</code> <code>\bool_lazy_all:nTF { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ... {⟨boolexpr_N⟩} } {⟨true code⟩} {⟨false code⟩}</code>
--	---

Implements the “And” operation on the *⟨boolean expressions⟩*, hence is `true` if all of them are `true` and `false` if any of them is `false`. Contrarily to the infix operator `&&`, only the *⟨boolean expressions⟩* which are needed to determine the result of `\bool_lazy_all:nTF` are evaluated. See also `\bool_lazy_and:nnTF` when there are only two *⟨boolean expressions⟩*.

<code>\bool_lazy_and_p:nn</code> ★ <code>\bool_lazy_and:nnTF</code> ★ <hr/> New: 2015-11-15 Updated: 2017-07-15	<code>\bool_lazy_and_p:nn {⟨boolexpr₁⟩} {⟨boolexpr₂⟩}</code> <code>\bool_lazy_and:nnTF {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} {⟨true code⟩} {⟨false code⟩}</code>
--	---

Implements the “And” operation between two boolean expressions, hence is `true` if both are `true`. Contrarily to the infix operator `&&`, the *⟨boolexpr₂⟩* is only evaluated if it is needed to determine the result of `\bool_lazy_and:nnTF`. See also `\bool_lazy_all:nTF` when there are more than two *⟨boolean expressions⟩*.

<code>\bool_lazy_any_p:n</code> ★ <code>\bool_lazy_any:nTF</code> ★ <hr/> New: 2015-11-15 Updated: 2017-07-15	<code>\bool_lazy_any_p:n { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ... {⟨boolexpr_N⟩} }</code> <code>\bool_lazy_any:nTF { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ... {⟨boolexpr_N⟩} } {⟨true code⟩} {⟨false code⟩}</code>
--	---

Implements the “Or” operation on the *⟨boolean expressions⟩*, hence is `true` if any of them is `true` and `false` if all of them are `false`. Contrarily to the infix operator `||`, only the *⟨boolean expressions⟩* which are needed to determine the result of `\bool_lazy_any:nTF` are evaluated. See also `\bool_lazy_or:nnTF` when there are only two *⟨boolean expressions⟩*.

<code>\bool_lazy_or_p:nn</code> ★ <code>\bool_lazy_or:nnTF</code> ★ <hr/> New: 2015-11-15 Updated: 2017-07-15	<code>\bool_lazy_or_p:nn {⟨boolexpr₁⟩} {⟨boolexpr₂⟩}</code> <code>\bool_lazy_or:nnTF {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} {⟨true code⟩} {⟨false code⟩}</code>
--	---

Implements the “Or” operation between two boolean expressions, hence is `true` if either one is `true`. Contrarily to the infix operator `||`, the *⟨boolexpr₂⟩* is only evaluated if it is needed to determine the result of `\bool_lazy_or:nnTF`. See also `\bool_lazy_any:nTF` when there are more than two *⟨boolean expressions⟩*.

<code>\bool_not_p:n</code> ★ <hr/> Updated: 2017-07-15	<code>\bool_not_p:n {⟨boolean expression⟩}</code> Function version of <code>!(⟨boolean expression⟩)</code> within a boolean expression.
---	--

<code>\bool_xor_p:nn</code> ☆
Updated: 2017-07-15

`\bool_xor_p:nn` $\{ \langle \text{boolean expr}_1 \rangle \} \{ \langle \text{boolean expr}_2 \rangle \}$

Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operator.

4 Logical loops

Loops using either boolean expressions or stored boolean values.

<code>\bool_do_until:Nn</code> ☆
<code>\bool_do_until:cn</code> ☆
Updated: 2017-07-15

`\bool_do_until:Nn` $\langle \text{boolean} \rangle \{ \langle \text{code} \rangle \}$

Places the $\langle \text{code} \rangle$ in the input stream for T_EX to process, and then checks the logical value of the $\langle \text{boolean} \rangle$. If it is **false** then the $\langle \text{code} \rangle$ is inserted into the input stream again and the process loops until the $\langle \text{boolean} \rangle$ is **true**.

<code>\bool_do_while:Nn</code> ☆
<code>\bool_do_while:cn</code> ☆
Updated: 2017-07-15

`\bool_do_while:Nn` $\langle \text{boolean} \rangle \{ \langle \text{code} \rangle \}$

Places the $\langle \text{code} \rangle$ in the input stream for T_EX to process, and then checks the logical value of the $\langle \text{boolean} \rangle$. If it is **true** then the $\langle \text{code} \rangle$ is inserted into the input stream again and the process loops until the $\langle \text{boolean} \rangle$ is **false**.

<code>\bool_until_do:Nn</code> ☆
<code>\bool_until_do:cn</code> ☆
Updated: 2017-07-15

`\bool_until_do:Nn` $\langle \text{boolean} \rangle \{ \langle \text{code} \rangle \}$

This function firsts checks the logical value of the $\langle \text{boolean} \rangle$. If it is **false** the $\langle \text{code} \rangle$ is placed in the input stream and expanded. After the completion of the $\langle \text{code} \rangle$ the truth of the $\langle \text{boolean} \rangle$ is re-evaluated. The process then loops until the $\langle \text{boolean} \rangle$ is **true**.

<code>\bool_while_do:Nn</code> ☆
<code>\bool_while_do:cn</code> ☆
Updated: 2017-07-15

`\bool_while_do:Nn` $\langle \text{boolean} \rangle \{ \langle \text{code} \rangle \}$

This function firsts checks the logical value of the $\langle \text{boolean} \rangle$. If it is **true** the $\langle \text{code} \rangle$ is placed in the input stream and expanded. After the completion of the $\langle \text{code} \rangle$ the truth of the $\langle \text{boolean} \rangle$ is re-evaluated. The process then loops until the $\langle \text{boolean} \rangle$ is **false**.

<code>\bool_do_until:nn</code> ☆
Updated: 2017-07-15

`\bool_do_until:nn` $\{ \langle \text{boolean expression} \rangle \} \{ \langle \text{code} \rangle \}$

Places the $\langle \text{code} \rangle$ in the input stream for T_EX to process, and then checks the logical value of the $\langle \text{boolean expression} \rangle$ as described for `\bool_if:nTF`. If it is **false** then the $\langle \text{code} \rangle$ is inserted into the input stream again and the process loops until the $\langle \text{boolean expression} \rangle$ evaluates to **true**.

<code>\bool_do_while:nn</code> ☆
Updated: 2017-07-15

`\bool_do_while:nn` $\{ \langle \text{boolean expression} \rangle \} \{ \langle \text{code} \rangle \}$

Places the $\langle \text{code} \rangle$ in the input stream for T_EX to process, and then checks the logical value of the $\langle \text{boolean expression} \rangle$ as described for `\bool_if:nTF`. If it is **true** then the $\langle \text{code} \rangle$ is inserted into the input stream again and the process loops until the $\langle \text{boolean expression} \rangle$ evaluates to **false**.

<code>\bool_until_do:nn</code> ☆
Updated: 2017-07-15

`\bool_until_do:nn` $\{ \langle \text{boolean expression} \rangle \} \{ \langle \text{code} \rangle \}$

This function firsts checks the logical value of the $\langle \text{boolean expression} \rangle$ (as described for `\bool_if:nTF`). If it is **false** the $\langle \text{code} \rangle$ is placed in the input stream and expanded. After the completion of the $\langle \text{code} \rangle$ the truth of the $\langle \text{boolean expression} \rangle$ is re-evaluated. The process then loops until the $\langle \text{boolean expression} \rangle$ is **true**.

`\bool_while_do:nn` ★

Updated: 2017-07-15

`\bool_while_do:nn` $\{\langle\textit{boolean expression}\rangle\}$ $\{\langle\textit{code}\rangle\}$

This function firsts checks the logical value of the $\langle\textit{boolean expression}\rangle$ (as described for `\bool_if:nTF`). If it is `true` the $\langle\textit{code}\rangle$ is placed in the input stream and expanded. After the completion of the $\langle\textit{code}\rangle$ the truth of the $\langle\textit{boolean expression}\rangle$ is re-evaluated. The process then loops until the $\langle\textit{boolean expression}\rangle$ is `false`.

5 Producing multiple copies

`\prg_replicate:nn` ★

Updated: 2011-07-04

`\prg_replicate:nn` $\{\langle\textit{integer expression}\rangle\}$ $\{\langle\textit{tokens}\rangle\}$

Evaluates the $\langle\textit{integer expression}\rangle$ (which should be zero or positive) and creates the resulting number of copies of the $\langle\textit{tokens}\rangle$. The function is both expandable and safe for nesting. It yields its result after two expansion steps.

6 Detecting T_EX's mode

`\mode_if_horizontal_p:` ★

`\mode_if_horizontal:TF` ★

`\mode_if_horizontal_p:`

`\mode_if_horizontal:TF` $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$

Detects if T_EX is currently in horizontal mode.

`\mode_if_inner_p:` ★

`\mode_if_inner:TF` ★

`\mode_if_inner_p:`

`\mode_if_inner:TF` $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$

Detects if T_EX is currently in inner mode.

`\mode_if_math_p:` ★

`\mode_if_math:TF` ★

`\mode_if_math:TF` $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$

Detects if T_EX is currently in maths mode.

Updated: 2011-09-05

`\mode_if_vertical_p:` ★

`\mode_if_vertical:TF` ★

`\mode_if_vertical_p:`

`\mode_if_vertical:TF` $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$

Detects if T_EX is currently in vertical mode.

7 Primitive conditionals

`\if_predicate:w` ★

`\if_predicate:w` $\langle\textit{predicate}\rangle$ $\langle\textit{true code}\rangle$ `\else:` $\langle\textit{false code}\rangle$ `\fi:`

This function takes a predicate function and branches according to the result. (In practice this function would also accept a single boolean variable in place of the $\langle\textit{predicate}\rangle$ but to make the coding clearer this should be done through `\if_bool:N`.)

`\if_bool:N` ★

`\if_bool:N` $\langle\textit{boolean}\rangle$ $\langle\textit{true code}\rangle$ `\else:` $\langle\textit{false code}\rangle$ `\fi:`

This function takes a boolean variable and branches according to the result.

8 Nestable recursions and mappings

There are a number of places where recursion or mapping constructs are used in `expl3`. At a low-level, these typically require insertion of tokens at the end of the content to allow “clean up”. To support such mappings in a nestable form, the following functions are provided.

<code>\prg_break_point:Nn</code> ★	<code>\prg_break_point:Nn \<type>_map_break: {\<code>}</code>
New: 2018-03-26	Used to mark the end of a recursion or mapping: the functions <code>\<type>_map_break:</code> and <code>\<type>_map_break:n</code> use this to break out of the loop (see <code>\prg_map_break:Nn</code> for how to set these up). After the loop ends, the <code>\<code></code> is inserted into the input stream. This occurs even if the break functions are <i>not</i> applied: <code>\prg_break_point:Nn</code> is functionally-equivalent in these cases to <code>\use_ii:nn</code> .

<code>\prg_map_break:Nn</code> ★	<code>\prg_map_break:Nn \<type>_map_break: {\<user code>}</code>
New: 2018-03-26	... <code>\prg_break_point:Nn \<type>_map_break: {\<ending code>}</code> Breaks a recursion in mapping contexts, inserting in the input stream the <code>\<user code></code> after the <code>\<ending code></code> for the loop. The function breaks loops, inserting their <code>\<ending code></code> , until reaching a loop with the same <code>\<type></code> as its first argument. This <code>\<type>_map_break:</code> argument must be defined; it is simply used as a recognizable marker for the <code>\<type></code> .

For types with mappings defined in the kernel, `\<type>_map_break:` and `\<type>_map_break:n` are defined as `\prg_map_break:Nn \<type>_map_break: {}` and the same with `{}` omitted.

8.1 Simple mappings

In addition to the more complex mappings above, non-nestable mappings are used in a number of locations and support is provided for these.

<code>\prg_break_point:</code> ★	This copy of <code>\prg_do_nothing:</code> is used to mark the end of a fast short-term recursion: the function <code>\prg_break:n</code> uses this to break out of the loop.
New: 2018-03-27	
<code>\prg_break:</code> ★	<code>\prg_break:n {\<code>} ... \prg_break_point:</code>
<code>\prg_break:n</code> ★	Breaks a recursion which has no <code>\<ending code></code> and which is not a user-breakable mapping (see for instance <code>\prop_get:Nn</code>), and inserts the <code>\<code></code> in the input stream.
New: 2018-03-27	

9 Internal programming functions

<code>\group_align_safe_begin:</code>	★	<code>\group_align_safe_begin:</code>
<code>\group_align_safe_end:</code>	★	<code>...</code>
		<code>\group_align_safe_end:</code>

Updated: 2011-08-11

These functions are used to enclose material in a \TeX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the `&` token inside `\halign`. This is necessary to allow grabbing of tokens for testing purposes, as \TeX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` would result in a forbidden comparison of the internal `\endtemplate` token, yielding a fatal error. Each `\group_align_safe_begin:` must be matched by a `\group_align_safe_end:`, although this does not have to occur within the same function.

Part XIII

The l3sys package: System/runtime functions

1 The name of the job

`\c_sys_jobname_str`

New: 2015-09-19

Constant that gets the “job name” assigned when T_EX starts.

T_EXhackers note: This copies the contents of the primitive `\jobname`. It is a constant that is set by T_EX and should not be overwritten by the package.

2 Date and time

`\c_sys_minute_int`
`\c_sys_hour_int`
`\c_sys_day_int`
`\c_sys_month_int`
`\c_sys_year_int`

New: 2015-09-22

The date and time at which the current job was started: these are all reported as integers.

T_EXhackers note: Whilst the underlying primitives can be altered by the user, this interface to the time and date is intended to be the “real” values.

3 Engine

`\sys_if_engine luatex_p:` ★
`\sys_if_engine luatex:` *TF* ★
`\sys_if_engine pdftex_p:` ★
`\sys_if_engine pdftex:` *TF* ★
`\sys_if_engine ptex_p:` ★
`\sys_if_engine ptex:` *TF* ★
`\sys_if_engine uptex_p:` ★
`\sys_if_engine uptex:` *TF* ★
`\sys_if_engine xetex_p:` ★
`\sys_if_engine xetex:` *TF* ★

New: 2015-09-07

`\sys_if_engine pdftex:TF` *{(true code)}* *{(false code)}*

Conditionals which allow engine-specific code to be used. The names follow naturally from those of the engine binaries: note that the (u)pt_EX tests are for ε -pT_EX and ε -upT_EX as expl3 requires the ε -T_EX extensions. Each conditional is true for *exactly one* supported engine. In particular, `\sys_if_engine ptex_p:` is true for ε -pT_EX but false for ε -upT_EX.

`\c_sys_engine_str`

New: 2015-09-19

The current engine given as a lower case string: one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

4 Output format

<code>\sys_if_output_dvi_p:</code>	★	<code>\sys_if_output_dvi:TF</code>	<code>{\true code}</code>	<code>{\false code}</code>
<code>\sys_if_output_dvi:</code>	<u>TF</u> ★	Conditionals which give the current output mode the TeX run is operating in. This is always one of two outcomes, DVI mode or PDF mode. The two sets of conditionals are thus complementary and are both provided to allow the programmer to emphasise the most appropriate case.		
<code>\sys_if_output_pdf_p:</code>	★			
<code>\sys_if_output_pdf:</code>	<u>TF</u> ★			
<hr/>				
New: 2015-09-19				

<code>\c_sys_output_str</code>	The current output mode given as a lower case string: one of <code>dvi</code> or <code>pdf</code> .
New: 2015-09-19	

Part XIV

The l3clist package

Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of the list. The features of this data type largely overlap with those of the sequence data type, for instance applying a function to every item, removing duplicate items, extracting a given item, using the comma list with specified separators, and so on.

Several items can be added at once. To ease input of comma lists from data provided by a user outside an `\ExplSyntaxOn ... \ExplSyntaxOff` block, spaces are removed from both sides of each comma-delimited argument upon input. Blank arguments are ignored, to allow for trailing commas or repeated commas (which may otherwise arise when concatenating comma lists “by hand”). In addition, a set of braces is removed if the result of space-trimming is braced: this allows the storage of any item in a comma list. For instance,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~a~ , ~{b}~ , c~\d }
\clist_put_right:Nn \l_my_clist { ~{e~} , , {{f}} , }
```

results in `\l_my_clist` containing `a,b,c~\d,{e~},{{f}}` namely the five items `a`, `b`, `c~\d`, `e~` and `{f}`. Comma lists normally do not contain empty items so the following gives an empty comma list:

```
\clist_clear_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

and it leaves `true` in the input stream. To include an “unsafe” item (empty, or one that contains a comma, or starts or ends with a space, or is a single brace group), surround it with braces.

Comma lists are particularly useful when implementing user interfaces, or when interfacing with L^AT_EX 2_ε or other code that expects or provides comma list data. Almost all operations on comma lists are noticeably slower than those on sequences so converting the data to sequences using `\seq_set_from_clist:Nn` (see l3seq) may be advisable if speed is important. The exception is that `\clist_if_in:NnTF` and `\clist_remove_duplicates:N` may be faster than their sequence analogues for large lists. However, these functions work slowly for “unsafe” items that must be braced, and may produce errors when their argument contains `{`, `}` or `#` (assuming the usual T_EX category codes apply). In addition, comma lists cannot store quarks `\q_mark` or `\q_stop`. The sequence data type should thus certainly be preferred to comma lists to store such items.

1 Creating and initialising comma lists

<code>\clist_new:N</code>	<code>\clist_new:N <comma list></code>
<code>\clist_new:c</code>	

Creates a new *<comma list>* or raises an error if the name is already taken. The declaration is global. The *<comma list>* initially contains no items.

<hr/> \clist_const:Nn \clist_const:(Nx cn cx) <hr/> New: 2014-07-05	\clist_const:Nn <clist var> {{<comma list>}} Creates a new constant <clist var> or raises an error if the name is already taken. The value of the <clist var> is set globally to the <comma list>.
<hr/> \clist_clear:N \clist_clear:c \clist_gclear:N \clist_gclear:c <hr/>	\clist_clear:N <comma list> Clears all items from the <comma list>.
<hr/> \clist_clear_new:N \clist_clear_new:c \clist_gclear_new:N \clist_gclear_new:c <hr/>	\clist_clear_new:N <comma list> Ensures that the <comma list> exists globally by applying \clist_new:N if necessary, then applies \clist_(g)clear:N to leave the list empty.
<hr/> \clist_set_eq:NN \clist_set_eq:(cN Nc cc) \clist_gset_eq:NN \clist_gset_eq:(cN Nc cc) <hr/>	\clist_set_eq:NN <comma list ₁ > <comma list ₂ > Sets the content of <comma list ₁ > equal to that of <comma list ₂ >.
<hr/> \clist_set_from_seq:NN \clist_set_from_seq:(cN Nc cc) \clist_gset_from_seq:NN \clist_gset_from_seq:(cN Nc cc) <hr/> New: 2014-07-17	\clist_set_from_seq:NN <comma list> <sequence> Converts the data in the <sequence> into a <comma list>: the original <sequence> is unchanged. Items which contain either spaces or commas are surrounded by braces.
<hr/> \clist_concat:NNN \clist_concat:ccc \clist_gconcat:NNN \clist_gconcat:ccc <hr/>	\clist_concat:NNN <comma list ₁ > <comma list ₂ > <comma list ₃ > Concatenates the content of <comma list ₂ > and <comma list ₃ > together and saves the result in <comma list ₁ >. The items in <comma list ₂ > are placed at the left side of the new comma list.
<hr/> \clist_if_exist_p:N ★ \clist_if_exist_p:c ★ \clist_if_exist:NTF ★ \clist_if_exist:cTF ★ <hr/> New: 2012-03-03	\clist_if_exist_p:N <comma list> \clist_if_exist:NTF <comma list> {{<true code>}} {{<false code>}} Tests whether the <comma list> is currently defined. This does not check that the <comma list> really is a comma list.

2 Adding data to comma lists

<code>\clist_set:Nn</code>	<code>\clist_set:Nn <comma list> {\<item_1>, ..., \<item_n>}</code>
<code>\clist_set:(NV No Nx cn cV co cx)</code>	
<code>\clist_gset:Nn</code>	
<code>\clist_gset:(NV No Nx cn cV co cx)</code>	

New: 2011-09-06

Sets $\langle comma list \rangle$ to contain the $\langle items \rangle$, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some $\langle tokens \rangle$ as a single $\langle item \rangle$ even if the $\langle tokens \rangle$ contain commas or spaces, add a set of braces: `\clist_set:Nn <comma list> { {\<tokens>} }`.

<code>\clist_put_left:Nn</code>	<code>\clist_put_left:Nn <comma list> {\<item_1>, ..., \<item_n>}</code>
<code>\clist_put_left:(NV No Nx cn cV co cx)</code>	
<code>\clist_gput_left:Nn</code>	
<code>\clist_gput_left:(NV No Nx cn cV co cx)</code>	

Updated: 2011-09-05

Appends the $\langle items \rangle$ to the left of the $\langle comma list \rangle$. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some $\langle tokens \rangle$ as a single $\langle item \rangle$ even if the $\langle tokens \rangle$ contain commas or spaces, add a set of braces: `\clist_put_left:Nn <comma list> { {\<tokens>} }`.

<code>\clist_put_right:Nn</code>	<code>\clist_put_right:Nn <comma list> {\<item_1>, ..., \<item_n>}</code>
<code>\clist_put_right:(NV No Nx cn cV co cx)</code>	
<code>\clist_gput_right:Nn</code>	
<code>\clist_gput_right:(NV No Nx cn cV co cx)</code>	

Updated: 2011-09-05

Appends the $\langle items \rangle$ to the right of the $\langle comma list \rangle$. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some $\langle tokens \rangle$ as a single $\langle item \rangle$ even if the $\langle tokens \rangle$ contain commas or spaces, add a set of braces: `\clist_put_right:Nn <comma list> { {\<tokens>} }`.

3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

<code>\clist_remove_duplicates:N</code>	<code>\clist_remove_duplicates:N <comma list></code>
<code>\clist_remove_duplicates:c</code>	
<code>\clist_gremove_duplicates:N</code>	
<code>\clist_gremove_duplicates:c</code>	

Removes duplicate items from the $\langle comma list \rangle$, leaving the left most copy of each item in the $\langle comma list \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

TeXhackers note: This function iterates through every item in the $\langle comma list \rangle$ and does a comparison with the $\langle items \rangle$ already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the $\langle comma list \rangle$ contains `{`, `}`, or `#` (assuming the usual TeX category codes apply).

<code>\clist_remove_all:Nn</code>	<code>\clist_remove_all:Nn <comma list> {<item>}</code>
<code>\clist_remove_all:cn</code>	
<code>\clist_gremove_all:Nn</code>	
<code>\clist_gremove_all:cn</code>	

Removes every occurrence of $\langle item \rangle$ from the $\langle comma list \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

Updated: 2011-09-06

TeXhackers note: The function may fail if the $\langle item \rangle$ contains `{`, `}`, or `#` (assuming the usual TeX category codes apply).

<code>\clist_reverse:N</code>	<code>\clist_reverse:N <comma list></code>
<code>\clist_reverse:c</code>	
<code>\clist_greverse:N</code>	
<code>\clist_greverse:c</code>	

New: 2014-07-18

Reverses the order of items stored in the $\langle comma list \rangle$.

<code>\clist_reverse:n</code>	<code>\clist_reverse:n {<comma list>}</code>
-------------------------------	--

New: 2014-07-18

Leaves the items in the $\langle comma list \rangle$ in the input stream in reverse order. Contrarily to other what is done for other n-type $\langle comma list \rangle$ arguments, braces and spaces are preserved by this process.

TeXhackers note: The result is returned within `\unexpanded`, which means that the comma list does not expand further when appearing in an x-type argument expansion.

<code>\clist_sort:Nn</code>	<code>\clist_sort:Nn <clist var> {<comparison code>}</code>
<code>\clist_sort:cn</code>	
<code>\clist_gsort:Nn</code>	
<code>\clist_gsort:cn</code>	

New: 2017-02-06

Sorts the items in the $\langle clist var \rangle$ according to the $\langle comparison code \rangle$, and assigns the result to $\langle clist var \rangle$. The details of sorting comparison are described in Section 1.

4 Comma list conditionals

<code>\clist_if_empty_p:N</code> ★	<code>\clist_if_empty_p:N <comma list></code>
<code>\clist_if_empty_p:c</code> ★	<code>\clist_if_empty_p:NTF <comma list> {<true code>} {<false code>}</code>
<code>\clist_if_empty:NTF</code> ★	
<code>\clist_if_empty:cTF</code> ★	

Tests if the $\langle comma list \rangle$ is empty (containing no items).

<code>\clist_if_empty_p:n</code> ★	<code>\clist_if_empty_p:n {⟨comma list⟩}</code>
<code>\clist_if_empty:nTF</code> ★	<code>\clist_if_empty:nTF {⟨comma list⟩} {⟨true code⟩} {⟨false code⟩}</code>

New: 2014-07-05

Tests if the $\langle comma list \rangle$ is empty (containing no items). The rules for space trimming are as for other n -type comma-list functions, hence the comma list $\{\sim, \sim, \sim\}$ (without outer braces) is empty, while $\{\sim, \{\}, \}$ (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

<code>\clist_if_in:NnTF</code>	<code>\clist_if_in:NnTF ⟨comma list⟩ {⟨item⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\clist_if_in:(NV No cn cV co)TF</code>	
<code>\clist_if_in:nnTF</code>	
<code>\clist_if_in:(nV no)TF</code>	

Updated: 2011-09-06

Tests if the $\langle item \rangle$ is present in the $\langle comma list \rangle$. In the case of an n -type $\langle comma list \rangle$, the usual rules of space trimming and brace stripping apply. Hence,

`\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}`

yields `true`.

T_EXhackers note: The function may fail if the $\langle item \rangle$ contains $\{$, $\}$, or $\#$ (assuming the usual T_EX category codes apply).

5 Mapping to comma lists

The functions described in this section apply a specified function to each item of a comma list.

When the comma list is given explicitly, as an n -type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is $\{a_{\square},_{\square}\{b\}_{\square},_{\square},\{\},_{\square}\{c\},\}$ then the arguments passed to the mapped function are ‘ a ’, ‘ b ’, an empty argument, and ‘ c ’.

When the comma list is given as an N -type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using n -type comma lists.

<code>\clist_map_function:NN</code> ★	<code>\clist_map_function:NN ⟨comma list⟩ ⟨function⟩</code>
---------------------------------------	---

`\clist_map_function:cN` ★

`\clist_map_function:nN` ★

Updated: 2012-06-29

Applies $\langle function \rangle$ to every $\langle item \rangle$ stored in the $\langle comma list \rangle$. The $\langle function \rangle$ receives one argument for each iteration. The $\langle items \rangle$ are returned from left to right. The function `\clist_map_inline:Nn` is in general more efficient than `\clist_map_function:NN`.

<code>\clist_map_inline:Nn</code>	<code>\clist_map_inline:Nn ⟨comma list⟩ {⟨inline function⟩}</code>
-----------------------------------	--

`\clist_map_inline:cn`

`\clist_map_inline:nn`

Updated: 2012-06-29

Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle comma list \rangle$. The $\langle inline function \rangle$ should consist of code which receives the $\langle item \rangle$ as $\#1$. The $\langle items \rangle$ are returned from left to right.

<code>\clist_map_variable:NNn</code>	<code>\clist_map_variable:NNn <comma list> <variable> {<code>}</code>
<code>\clist_map_variable:cNn</code>	Stores each <i><item></i> of the <i><comma list></i> in turn in the (token list) <i><variable></i> and applies the <i><code></i> . The <i><code></i> will usually make use of the <i><variable></i> , but this is not enforced.
<code>\clist_map_variable:nNn</code>	The assignments to the <i><variable></i> are local. The <i><items></i> are returned from left to right.
Updated: 2012-06-29	

<code>\clist_map_break:</code> ☆	<code>\clist_map_break:</code>
Updated: 2012-06-29	Used to terminate a <code>\clist_map...</code> function before all entries in the <i><comma list></i> have been processed. This normally takes place within a conditional statement, for example

```

\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break: }
  {
    % Do something useful
  }
}

```

Use outside of a `\clist_map...` scenario leads to low level \TeX errors.

\TeX hackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

<code>\clist_map_break:n</code> ☆	<code>\clist_map_break:n {<code>}</code>
Updated: 2012-06-29	Used to terminate a <code>\clist_map...</code> function before all entries in the <i><comma list></i> have been processed, inserting the <i><code></i> after the mapping has ended. This normally takes place within a conditional statement, for example

```

\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \clist_map_break:n { <code> } }
  {
    % Do something useful
  }
}

```

Use outside of a `\clist_map...` scenario leads to low level \TeX errors.

\TeX hackers note: When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

<code>\clist_count:N</code> ☆	<code>\clist_count:N <comma list></code>
<code>\clist_count:c</code> ☆	Leaves the number of items in the <i><comma list></i> in the input stream as an <i><integer denotation></i> . The total number of items in a <i><comma list></i> includes those which are duplicates, <i>i.e.</i> every item in a <i><comma list></i> is counted.
<code>\clist_count:n</code> ☆	
New: 2012-07-13	

6 Using the content of comma lists directly

<code>\clist_use:Nnnn</code> ★ <code>\clist_use:cnnn</code> ★	<code>\clist_use:Nnnn <clist var> {<separator between two>}</code> <code>{<separator between more than two>} {<separator between final two>}</code>
--	--

New: 2013-05-26

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the appropriate $\langle\textit{separator}\rangle$ between the items. Namely, if the comma list has more than two items, the $\langle\textit{separator between more than two}\rangle$ is placed between each pair of items except the last, for which the $\langle\textit{separator between final two}\rangle$ is used. If the comma list has exactly two items, then they are placed in the input stream separated by the $\langle\textit{separator between two}\rangle$. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle\textit{items}\rangle$ do not expand further when appearing in an x-type argument expansion.

<code>\clist_use:Nn</code> ★ <code>\clist_use:cn</code> ★	<code>\clist_use:Nn <clist var> {<separator>}</code>
--	--

New: 2013-05-26

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the $\langle\textit{separator}\rangle$ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle\textit{items}\rangle$ do not expand further when appearing in an x-type argument expansion.

7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

<u>\clist_get:NN</u> <u>\clist_get:cN</u> <hr/> Updated: 2012-05-14	<u>\clist_get:NN</u> $\langle comma list \rangle$ $\langle token list variable \rangle$ Stores the left-most item from a $\langle comma list \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle comma list \rangle$. The $\langle token list variable \rangle$ is assigned locally. If the $\langle comma list \rangle$ is empty the $\langle token list variable \rangle$ is set to the marker value $\backslash q_no_value$.
---	--

<u>\clist_get:NNTF</u> <u>\clist_get:cNTF</u> <hr/> New: 2012-05-14	<u>\clist_get:NNTF</u> $\langle comma list \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$ If the $\langle comma list \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle comma list \rangle$ is non-empty, stores the top item from the $\langle comma list \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle comma list \rangle$. The $\langle token list variable \rangle$ is assigned locally.
---	--

<u>\clist_pop:NN</u> <u>\clist_pop:cN</u> <hr/> Updated: 2011-09-06	<u>\clist_pop:NN</u> $\langle comma list \rangle$ $\langle token list variable \rangle$ Pops the left-most item from a $\langle comma list \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the comma list and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally.
---	---

<u>\clist_gpop:NN</u> <u>\clist_gpop:cN</u>	<u>\clist_gpop:NN</u> $\langle comma list \rangle$ $\langle token list variable \rangle$ Pops the left-most item from a $\langle comma list \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the comma list and stores it in the $\langle token list variable \rangle$. The $\langle comma list \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local.
--	---

<u>\clist_pop:NNTF</u> <u>\clist_pop:cNTF</u> <hr/> New: 2012-05-14	<u>\clist_pop:NNTF</u> $\langle comma list \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$ If the $\langle comma list \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle comma list \rangle$ is non-empty, pops the top item from the $\langle comma list \rangle$ in the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the $\langle comma list \rangle$. Both the $\langle comma list \rangle$ and the $\langle token list variable \rangle$ are assigned locally.
---	--

<u>\clist_gpop:NNTF</u> <u>\clist_gpop:cNTF</u> <hr/> New: 2012-05-14	<u>\clist_gpop:NNTF</u> $\langle comma list \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$ If the $\langle comma list \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle comma list \rangle$ is non-empty, pops the top item from the $\langle comma list \rangle$ in the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the $\langle comma list \rangle$. The $\langle comma list \rangle$ is modified globally, while the $\langle token list variable \rangle$ is assigned locally.
---	---

<u>\clist_push:Nn</u> <u>\clist_push:(NV No Nx cn cV co cx)</u> <u>\clist_gpush:Nn</u> <u>\clist_gpush:(NV No Nx cn cV co cx)</u>	<u>\clist_push:Nn</u> $\langle comma list \rangle$ $\{\langle items \rangle\}$
--	--

Adds the $\{\langle items \rangle\}$ to the top of the $\langle comma list \rangle$. Spaces are removed from both sides of each item as for any n-type comma list.

8 Using a single item

<code>\clist_item:Nn</code> ★	<code>\clist_item:Nn <comma list> {<integer expression>}</code>
<code>\clist_item:cn</code> ★	
<code>\clist_item:nn</code> ★	Indexing items in the <i><comma list></i> from 1 at the top (left), this function evaluates the <i><integer expression></i> and leaves the appropriate item from the comma list in the input stream. If the <i><integer expression></i> is negative, indexing occurs from the bottom (right) of the comma list. When the <i><integer expression></i> is larger than the number of items in the <i><comma list></i> (as calculated by <code>\clist_count:N</code>) then the function expands to nothing.
New: 2014-07-17	

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an *x*-type argument expansion.

9 Viewing comma lists

<code>\clist_show:N</code>	<code>\clist_show:N <comma list></code>
<code>\clist_show:c</code>	Displays the entries in the <i><comma list></i> in the terminal.
Updated: 2015-08-03	
<code>\clist_show:n</code>	<code>\clist_show:n {<tokens>}</code>
<code>\clist_show:c</code>	Displays the entries in the comma list in the terminal.
Updated: 2013-08-03	
<code>\clist_log:N</code>	<code>\clist_log:N <comma list></code>
<code>\clist_log:c</code>	Writes the entries in the <i><comma list></i> in the log file. See also <code>\clist_show:N</code> which displays the result in the terminal.
New: 2014-08-22	
Updated: 2015-08-03	
<code>\clist_log:n</code>	<code>\clist_log:n {<tokens>}</code>
<code>\clist_log:c</code>	Writes the entries in the comma list in the log file. See also <code>\clist_show:n</code> which displays the result in the terminal.
New: 2014-08-22	

10 Constant and scratch comma lists

<code>\c_empty_clist</code>	Constant that is always empty.
New: 2012-07-02	
<code>\l_tmpa_clist</code>	Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_clist</code>	
New: 2011-09-06	
<code>\g_tmpa_clist</code>	Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_clist</code>	
New: 2011-09-06	

Part XV

The l3token package

Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in T_EX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we describe here can be used on control sequences, as those are tokens as well.

It is important to distinguish two aspects of a token: its “shape” (for lack of a better word), which affects the matching of delimited arguments and the comparison of token lists containing this token, and its “meaning”, which affects whether the token expands or what operation it performs. One can have tokens of different shapes with the same meaning, but not the converse.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of T_EX, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence behave identically in many situations. However, T_EX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below takes everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

A list of all possible shapes and a list of all possible meanings are given in section 8.

1 Creating character tokens

```
\char_set_active_eq:NN
\char_set_active_eq:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc
```

Updated: 2015-11-12

```
\char_set_active_eq:NN <char> <function>
```

Sets the behaviour of the `<char>` in situations where it is active (category code 13) to be equivalent to that of the `<function>`. The category code of the `<char>` is *unchanged* by this process. The `<function>` may itself be an active character.

```
\char_set_active_eq:nN
\char_set_active_eq:nc
\char_gset_active_eq:nN
\char_gset_active_eq:nc
```

New: 2015-11-12

```
\char_set_active_eq:nN {<integer expression>} <function>
```

Sets the behaviour of the `<char>` which has character code as given by the `<integer expression>` in situations where it is active (category code 13) to be equivalent to that of the `<function>`. The category code of the `<char>` is *unchanged* by this process. The `<function>` may itself be an active character.

<hr/> <code>\char_generate:nn</code> ★ <hr/>	<code>\char_generate:nn {<charcode>} {<catcode>}</code>
New: 2015-09-09 Updated: 2018-04-19 <hr/>	<p>Generates a character token of the given $\langle charcode \rangle$ and $\langle catcode \rangle$ (both of which may be integer expressions). The $\langle catcode \rangle$ may be one of</p> <ul style="list-style-type: none"> • 1 (begin group) • 2 (end group) • 3 (math toggle) • 4 (alignment) • 6 (parameter) • 7 (math superscript) • 8 (math subscript) • 11 (letter) • 12 (other) • 13 (active) (not \XeTeX) <p>and other values raise an error.</p> <p>The $\langle charcode \rangle$ may be any one valid for the engine in use. Note however that for \XeTeX releases prior to 0.99992 only the 8-bit range (0 to 255) is accepted due to engine limitations. Note also that \XeTeX cannot currently generate active character tokens.</p>
<hr/> <code>\c_catcode_other_space_tl</code> <hr/>	Token list containing one character with category code 12, (“other”), and character code 32 (space).
New: 2011-09-05 <hr/>	

2 Manipulating and interrogating character tokens

<code>\char_set_catcode_escape:N</code>	<code>\char_set_catcode_letter:N</code> $\langle character \rangle$
<code>\char_set_catcode_group_begin:N</code>	
<code>\char_set_catcode_group_end:N</code>	
<code>\char_set_catcode_math_toggle:N</code>	
<code>\char_set_catcode_alignment:N</code>	
<code>\char_set_catcode_end_line:N</code>	
<code>\char_set_catcode_parameter:N</code>	
<code>\char_set_catcode_math_superscript:N</code>	
<code>\char_set_catcode_math_subscript:N</code>	
<code>\char_set_catcode_ignore:N</code>	
<code>\char_set_catcode_space:N</code>	
<code>\char_set_catcode_letter:N</code>	
<code>\char_set_catcode_other:N</code>	
<code>\char_set_catcode_active:N</code>	
<code>\char_set_catcode_comment:N</code>	
<code>\char_set_catcode_invalid:N</code>	

Updated: 2015-11-11

Sets the category code of the $\langle character \rangle$ to that indicated in the function name. Depending on the current category code of the $\langle token \rangle$ the escape token may also be needed:

`\char_set_catcode_other:N \%`

The assignment is local.

<code>\char_set_catcode_escape:n</code>	<code>\char_set_catcode_letter:n</code> $\{ \langle integer\ expression \rangle \}$
<code>\char_set_catcode_group_begin:n</code>	
<code>\char_set_catcode_group_end:n</code>	
<code>\char_set_catcode_math_toggle:n</code>	
<code>\char_set_catcode_alignment:n</code>	
<code>\char_set_catcode_end_line:n</code>	
<code>\char_set_catcode_parameter:n</code>	
<code>\char_set_catcode_math_superscript:n</code>	
<code>\char_set_catcode_math_subscript:n</code>	
<code>\char_set_catcode_ignore:n</code>	
<code>\char_set_catcode_space:n</code>	
<code>\char_set_catcode_letter:n</code>	
<code>\char_set_catcode_other:n</code>	
<code>\char_set_catcode_active:n</code>	
<code>\char_set_catcode_comment:n</code>	
<code>\char_set_catcode_invalid:n</code>	

Updated: 2015-11-11

Sets the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer\ expression \rangle$. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

<hr/> <code>\char_set_catcode:nn</code> <hr/>	<code>\char_set_catcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
<hr/> Updated: 2015-11-11 <hr/>	These functions set the category code of the <i>⟨character⟩</i> which has character code as given by the <i>⟨integer expression⟩</i> . The first <i>⟨integer expression⟩</i> is the character code and the second is the category code to apply. The setting applies within the current T _E X group. In general, the symbolic functions <code>\char_set_catcode_⟨type⟩</code> should be preferred, but there are cases where these lower-level functions may be useful.
<hr/> <code>\char_value_catcode:n</code> ★ <hr/>	<code>\char_value_catcode:n {⟨integer expression⟩}</code>
	Expands to the current category code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
<hr/> <code>\char_show_value_catcode:n</code> <hr/>	<code>\char_show_value_catcode:n {⟨integer expression⟩}</code>
	Displays the current category code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.
<hr/> <code>\char_set_lccode:nn</code> <hr/>	<code>\char_set_lccode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
<hr/> Updated: 2015-08-06 <hr/>	Sets up the behaviour of the <i>⟨character⟩</i> when found inside <code>\tl_lower_case:n</code> , such that <i>⟨character₁⟩</i> will be converted into <i>⟨character₂⟩</i> . The two <i>⟨characters⟩</i> may be specified using an <i>⟨integer expression⟩</i> for the character code concerned. This may include the T _E X ‘ <i>⟨character⟩</i> ’ method for converting a single character into its character code:
	<pre> \char_set_lccode:nn { ‘\A } { ‘\a } % Standard behaviour \char_set_lccode:nn { ‘\A } { ‘\A + 32 } \char_set_lccode:nn { 50 } { 60 } </pre>
	The setting applies within the current T _E X group.
<hr/> <code>\char_value_lccode:n</code> ★ <hr/>	<code>\char_value_lccode:n {⟨integer expression⟩}</code>
	Expands to the current lower case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
<hr/> <code>\char_show_value_lccode:n</code> <hr/>	<code>\char_show_value_lccode:n {⟨integer expression⟩}</code>
	Displays the current lower case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.
<hr/> <code>\char_set_uccode:nn</code> <hr/>	<code>\char_set_uccode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
<hr/> Updated: 2015-08-06 <hr/>	Sets up the behaviour of the <i>⟨character⟩</i> when found inside <code>\tl_upper_case:n</code> , such that <i>⟨character₁⟩</i> will be converted into <i>⟨character₂⟩</i> . The two <i>⟨characters⟩</i> may be specified using an <i>⟨integer expression⟩</i> for the character code concerned. This may include the T _E X ‘ <i>⟨character⟩</i> ’ method for converting a single character into its character code:
	<pre> \char_set_uccode:nn { ‘\a } { ‘\A } % Standard behaviour \char_set_uccode:nn { ‘\A } { ‘\A - 32 } \char_set_uccode:nn { 60 } { 50 } </pre>
	The setting applies within the current T _E X group.

<hr/> <hr/>	<hr/>
<code>\char_value_uccode:n</code> ★	<code>\char_value_uccode:n {\langle integer expression \rangle}</code>
	Expands to the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <hr/>	<hr/>
<code>\char_show_value_uccode:n</code>	<code>\char_show_value_uccode:n {\langle integer expression \rangle}</code>
	Displays the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <hr/>	<hr/>
<code>\char_set_mathcode:nn</code>	<code>\char_set_mathcode:nn {\langle intexpr_1 \rangle} {\langle intexpr_2 \rangle}</code>
Updated: 2015-08-06	This function sets up the math code of $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current T _E X group.
<hr/> <hr/>	<hr/>
<code>\char_value_mathcode:n</code> ★	<code>\char_value_mathcode:n {\langle integer expression \rangle}</code>
	Expands to the current math code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <hr/>	<hr/>
<code>\char_show_value_mathcode:n</code>	<code>\char_show_value_mathcode:n {\langle integer expression \rangle}</code>
	Displays the current math code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <hr/>	<hr/>
<code>\char_set_sfcode:nn</code>	<code>\char_set_sfcode:nn {\langle intexpr_1 \rangle} {\langle intexpr_2 \rangle}</code>
Updated: 2015-08-06	This function sets up the space factor for the $\langle character \rangle$. The $\langle character \rangle$ is specified as an $\langle integer expression \rangle$ which will be used as the character code of the relevant character. The setting applies within the current T _E X group.
<hr/> <hr/>	<hr/>
<code>\char_value_sfcode:n</code> ★	<code>\char_value_sfcode:n {\langle integer expression \rangle}</code>
	Expands to the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.
<hr/> <hr/>	<hr/>
<code>\char_show_value_sfcode:n</code>	<code>\char_show_value_sfcode:n {\langle integer expression \rangle}</code>
	Displays the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.
<hr/> <hr/>	<hr/>
<code>\l_char_active_seq</code>	Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category $\langle active \rangle$ (catcode 13). Each entry in the sequence consists of a single escaped token, for example <code>\~</code> . Active tokens should be added to the sequence when they are defined for general document use.
New: 2012-01-23 Updated: 2015-11-11	
<hr/> <hr/>	<hr/>
<code>\l_char_special_seq</code>	Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories $\langle letter \rangle$ (catcode 11) or $\langle other \rangle$ (catcode 12). Each entry in the sequence consists of a single escaped token, for example <code>\</code> for the backslash or <code>\{</code> for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.
New: 2012-01-23 Updated: 2015-11-11	
<hr/> <hr/>	<hr/>

3 Generic tokens

```
\c_group_begin_token
\c_group_end_token
\c_math_toggle_token
\c_alignment_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token
```

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

```
\c_catcode_letter_token
\c_catcode_other_token
```

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.

```
\c_catcode_active_tl
```

A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.

4 Converting tokens

```
\token_to_meaning:N ★
\token_to_meaning:c ★
```

`\token_to_meaning:N` $\langle token \rangle$

Inserts the current meaning of the $\langle token \rangle$ into the input stream as a series of characters of category code 12 (other). This is the primitive \TeX description of the $\langle token \rangle$, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` are described as **macros**.

\TeX hackers note: This is the \TeX primitive `\meaning`. The $\langle token \rangle$ can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal \TeX category codes apply) even though these are not valid \TeX -type arguments.

```
\token_to_str:N ★
\token_to_str:c ★
```

`\token_to_str:N` $\langle token \rangle$

Converts the given $\langle token \rangle$ into a series of characters with category code 12 (other). If the $\langle token \rangle$ is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the $\langle token \rangle$). This function requires only a single expansion.

\TeX hackers note: `\token_to_str:N` is the \TeX primitive `\string` renamed. The $\langle token \rangle$ can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal \TeX category codes apply) even though these are not valid \TeX -type arguments.

5 Token conditionals

<code>\token_if_group_begin_p:N</code>	★	<code>\token_if_group_begin_p:N</code>	$\langle token \rangle$
<code>\token_if_group_begin:NTF</code>	★	<code>\token_if_group_begin:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a begin group token (`{` when normal \TeX category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_group_end_p:N</code>	★	<code>\token_if_group_end_p:N</code>	$\langle token \rangle$
<code>\token_if_group_end:NTF</code>	★	<code>\token_if_group_end:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an end group token (`}` when normal \TeX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_math_toggle_p:N</code>	★	<code>\token_if_math_toggle_p:N</code>	$\langle token \rangle$
<code>\token_if_math_toggle:NTF</code>	★	<code>\token_if_math_toggle:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a math shift token (`$` when normal \TeX category codes are in force).

<code>\token_if_alignment_p:N</code>	★	<code>\token_if_alignment_p:N</code>	$\langle token \rangle$
<code>\token_if_alignment:NTF</code>	★	<code>\token_if_alignment:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an alignment token (`&` when normal \TeX category codes are in force).

<code>\token_if_parameter_p:N</code>	★	<code>\token_if_parameter_p:N</code>	$\langle token \rangle$
<code>\token_if_parameter:NTF</code>	★	<code>\token_if_parameter:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a macro parameter token (`#` when normal \TeX category codes are in force).

<code>\token_if_math_superscript_p:N</code>	★	<code>\token_if_math_superscript_p:N</code>	$\langle token \rangle$
<code>\token_if_math_superscript:NTF</code>	★	<code>\token_if_math_superscript:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a superscript token (`^` when normal \TeX category codes are in force).

<code>\token_if_math_subscript_p:N</code>	★	<code>\token_if_math_subscript_p:N</code>	$\langle token \rangle$
<code>\token_if_math_subscript:NTF</code>	★	<code>\token_if_math_subscript:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a subscript token (`_` when normal \TeX category codes are in force).

<code>\token_if_space_p:N</code>	★	<code>\token_if_space_p:N</code>	$\langle token \rangle$
<code>\token_if_space:NTF</code>	★	<code>\token_if_space:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_letter_p:N</code>	★	<code>\token_if_letter_p:N</code>	$\langle token \rangle$
<code>\token_if_letter:NTF</code>	★	<code>\token_if_letter:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of a letter token.

<code>\token_if_other_p:N</code>	★	<code>\token_if_other_p:N</code>	$\langle token \rangle$
<code>\token_if_other:NTF</code>	★	<code>\token_if_other:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an “other” token.

<code>\token_if_active_p:N</code>	★	<code>\token_if_active_p:N</code>	$\langle token \rangle$
<code>\token_if_active:NTF</code>	★	<code>\token_if_active:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if $\langle token \rangle$ has the category code of an active character.

<code>\token_if_eq_catcode_p:NN</code>	★	<code>\token_if_eq_catcode_p:NN</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$
<code>\token_if_eq_catcode:NNTF</code>	★	<code>\token_if_eq_catcode:NNTF</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the two $\langle tokens \rangle$ have the same category code.

<code>\token_if_eq_charcode_p:NN</code>	★	<code>\token_if_eq_charcode_p:NN</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$
<code>\token_if_eq_charcode:NNTF</code>	★	<code>\token_if_eq_charcode:NNTF</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the two $\langle tokens \rangle$ have the same character code.

<code>\token_if_eq_meaning_p:NN</code>	★	<code>\token_if_eq_meaning_p:NN</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$
<code>\token_if_eq_meaning:NNTF</code>	★	<code>\token_if_eq_meaning:NNTF</code>	$\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the two $\langle tokens \rangle$ have the same meaning when expanded.

<code>\token_if_macro_p:N</code>	★	<code>\token_if_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_macro:NTF</code>	★	<code>\token_if_macro:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2011-05-23

Tests if the $\langle token \rangle$ is a T_EX macro.

<code>\token_if_cs_p:N</code>	★	<code>\token_if_cs_p:N</code>	$\langle token \rangle$
<code>\token_if_cs:NTF</code>	★	<code>\token_if_cs:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle token \rangle$ is a control sequence.

<code>\token_if_expandable_p:N</code>	★	<code>\token_if_expandable_p:N</code>	$\langle token \rangle$
<code>\token_if_expandable:NTF</code>	★	<code>\token_if_expandable:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle token \rangle$ is expandable. This test returns $\langle false \rangle$ for an undefined token.

<code>\token_if_long_macro_p:N</code>	★	<code>\token_if_long_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_long_macro:NTF</code>	★	<code>\token_if_long_macro:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is a long macro.

<code>\token_if_protected_macro_p:N</code>	★	<code>\token_if_protected_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_protected_macro:NTF</code>	★	<code>\token_if_protected_macro:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is a protected macro: for a macro which is both protected and long this returns **false**.

<code>\token_if_protected_long_macro_p:N</code>	★	<code>\token_if_protected_long_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_protected_long_macro:NTF</code>	★	<code>\token_if_protected_long_macro:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is a protected long macro.

<code>\token_if_chardef_p:N</code>	★	<code>\token_if_chardef_p:N</code>	$\langle token \rangle$
<code>\token_if_chardef:NTF</code>	★	<code>\token_if_chardef:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a chardef.

TeXhackers note: Booleans, boxes and small integer constants are implemented as `\chardefs`.

<code>\token_if_mathchardef_p:N</code>	★	<code>\token_if_mathchardef_p:N</code>	$\langle token \rangle$
<code>\token_if_mathchardef:NTF</code>	★	<code>\token_if_mathchardef:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a mathchardef.

<code>\token_if_dim_register_p:N</code>	★	<code>\token_if_dim_register_p:N</code>	$\langle token \rangle$
<code>\token_if_dim_register:NTF</code>	★	<code>\token_if_dim_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a dimension register.

<code>\token_if_int_register_p:N</code>	★	<code>\token_if_int_register_p:N</code>	$\langle token \rangle$
<code>\token_if_int_register:NTF</code>	★	<code>\token_if_int_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a integer register.

TeXhackers note: Constant integers may be implemented as integer registers, `\chardefs`, or `\mathchardefs` depending on their value.

<code>\token_if_muskip_register_p:N</code>	★	<code>\token_if_muskip_register_p:N</code>	$\langle token \rangle$
<code>\token_if_muskip_register:NTF</code>	★	<code>\token_if_muskip_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

New: 2012-02-15

Tests if the $\langle token \rangle$ is defined to be a muskip register.

<code>\token_if_skip_register_p:N</code>	★	<code>\token_if_skip_register_p:N</code>	$\langle token \rangle$
<code>\token_if_skip_register:NTF</code>	★	<code>\token_if_skip_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a skip register.

<code>\token_if_toks_register_p:N</code>	★	<code>\token_if_toks_register_p:N</code>	$\langle token \rangle$
<code>\token_if_toks_register:NTF</code>	★	<code>\token_if_toks_register:NTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a toks register (not used by L^AT_EX3).

<code>\token_if_primitive_p:N</code>	★	<code>\token_if_primitive_p:N</code>	$\langle token \rangle$
<code>\token_if_primitive:NTF</code>	★	<code>\token_if_primitive:NTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2011-05-23

Tests if the $\langle token \rangle$ is an engine primitive.

6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the “peek” functions. The generic `\peek_after:Nw` is provided along with a family of predefined tests for common cases. As peeking ahead does *not* skip spaces the predefined tests include both a space-respecting and space-skipping version.

<code>\peek_after:Nw</code>	<code>\peek_after:Nw</code>	$\langle function \rangle$	$\langle token \rangle$
-----------------------------	-----------------------------	----------------------------	-------------------------

Locally sets the test variable `\l_peek_token` equal to $\langle token \rangle$ (as an implicit token, *not* as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ remains in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ here may be \sqcup , $\{$ or $\}$ (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

<code>\peek_gafter:Nw</code>	<code>\peek_gafter:Nw</code>	$\langle function \rangle$	$\langle token \rangle$
------------------------------	------------------------------	----------------------------	-------------------------

Globally sets the test variable `\g_peek_token` equal to $\langle token \rangle$ (as an implicit token, *not* as a token list), and then expands the $\langle function \rangle$. The $\langle token \rangle$ remains in the input stream as the next item after the $\langle function \rangle$. The $\langle token \rangle$ here may be \sqcup , $\{$ or $\}$ (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

<code>\l_peek_token</code>	Token set by <code>\peek_after:Nw</code> and available for testing as described above.
----------------------------	--

<code>\g_peek_token</code>	Token set by <code>\peek_gafter:Nw</code> and available for testing as described above.
----------------------------	---

<code>\peek_catcode:NTF</code>	<code>\peek_catcode:NTF</code>	$\langle test\ token \rangle$	$\{\langle true\ code \rangle\}$	$\{\langle false\ code \rangle\}$
--------------------------------	--------------------------------	-------------------------------	----------------------------------	-----------------------------------

Updated: 2012-12-20

Tests if the next $\langle token \rangle$ in the input stream has the same category code as the $\langle test\ token \rangle$ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the $\langle token \rangle$ is left in the input stream after the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ (as appropriate to the result of the test).

<code>\peek_catcode_ignore_spaces:NTF</code>	<code>\peek_catcode_ignore_spaces:NTF <test token> {(true code)} {(false code)}</code>
--	--

Updated: 2012-12-20

Tests if the next non-space *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<code>\peek_catcode_remove:NTF</code>	<code>\peek_catcode_remove:NTF <test token> {(true code)} {(false code)}</code>
---------------------------------------	---

Updated: 2012-12-20

Tests if the next *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<code>\peek_catcode_remove_ignore_spaces:NTF</code>	<code>\peek_catcode_remove_ignore_spaces:NTF <test token> {(true code)} {(false code)}</code>
---	---

Updated: 2012-12-20

Tests if the next non-space *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<code>\peek_charcode:NTF</code>	<code>\peek_charcode:NTF <test token> {(true code)} {(false code)}</code>
---------------------------------	---

Updated: 2012-12-20

Tests if the next *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<code>\peek_charcode_ignore_spaces:NTF</code>	<code>\peek_charcode_ignore_spaces:NTF <test token> {(true code)} {(false code)}</code>
---	---

Updated: 2012-12-20

Tests if the next non-space *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<code>\peek_charcode_remove:NTF</code>	<code>\peek_charcode_remove:NTF <test token> {(true code)} {(false code)}</code>
--	--

Updated: 2012-12-20

Tests if the next *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<u><code>\peek_charcode_remove_ignore_spaces:NTF</code></u>	<code>\peek_charcode_remove_ignore_spaces:NTF <test token> {<true code>} {<false code>}</code>
Updated: 2012-12-20	

Tests if the next non-space *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<u><code>\peek_meaning:NTF</code></u>	<code>\peek_meaning:NTF <test token> {<true code>} {<false code>}</code>
Updated: 2011-07-02	

Tests if the next *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<u><code>\peek_meaning_ignore_spaces:NTF</code></u>	<code>\peek_meaning_ignore_spaces:NTF <test token> {<true code>} {<false code>}</code>
Updated: 2012-12-05	

Tests if the next non-space *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<u><code>\peek_meaning_remove:NTF</code></u>	<code>\peek_meaning_remove:NTF <test token> {<true code>} {<false code>}</code>
Updated: 2011-07-02	

Tests if the next *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<u><code>\peek_meaning_remove_ignore_spaces:NTF</code></u>	<code>\peek_meaning_remove_ignore_spaces:NTF <test token> {<true code>} {<false code>}</code>
Updated: 2012-12-05	

Tests if the next non-space *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

7 Decomposing a macro definition

These functions decompose \TeX macros into their constituent parts: if the *<token>* passed is not a macro then no decomposition can occur. In the latter case, all three functions leave `\scan_stop:` in the input stream.

`\token_get_arg_spec:N` ★

`\token_get_arg_spec:N` $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function leaves the primitive \TeX argument specification in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

`\cs_set:Npn \next #1#2 { x #1 y #2 }`

leaves `#1#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` is left in the input stream.

\TeX hackers note: If the arg spec. contains the string `->`, then the `spec` function produces incorrect results.

`\token_get_replacement_spec:N` ★

`\token_get_replacement_spec:N` $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function leaves the replacement text in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

`\cs_set:Npn \next #1#2 { x #1~y #2 }`

leaves `x#1 y#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` is left in the input stream.

\TeX hackers note: If the arg spec. contains the string `->`, then the `spec` function produces incorrect results.

`\token_get_prefix_spec:N` ★

`\token_get_prefix_spec:N` $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function leaves the \TeX prefixes applicable in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token `\next` defined by

`\cs_set:Npn \next #1#2 { x #1~y #2 }`

leaves `\long` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` is left in the input stream

8 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on \TeX 's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, `\use:n` is a five-letter control sequence.

- An active character token, characterized by its character code (between 0 and 1114111 for LuaTeX and XeTeX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).⁴

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding `\the\font` results in a token that looks identical to the command that was used to select the current font (such as `\tenrm`) but it differs from it in shape.
- A “frozen” `\relax`, which differs from the primitive in shape (but has the same meaning), is inserted when the closing `\fi` of a conditional is encountered before the conditional is evaluated.
- Expanding `\noexpand⟨token⟩` (when the `⟨token⟩` is expandable) results in an internal token, displayed (temporarily) as `\notexpanded:⟨token⟩`, whose shape coincides with the `⟨token⟩` and whose meaning differs from `\relax`.
- An `\outer endtemplate:` can be encountered when peeking ahead at the next token; this expands to another internal token, `end of alignment template`.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the TeX primitive `\meaning`, together with their L^AT_EX3 names and most common example:

- 1 begin-group character (`group_begin`, often `{`),
- 2 end-group character (`group_end`, often `}`),
- 3 math shift character (`math_toggle`, often `$`),
- 4 alignment tab character (`alignment`, often `&`),
- 6 macro parameter character (`parameter`, often `#`),
- 7 superscript character (`math_superscript`, often `^`),
- 8 subscript character (`math_subscript`, often `_`),
- 10 blank space (`space`, often character code 32),
- 11 the letter (`letter`, such as `A`),
- 12 the character (`other`, such as `0`).

⁴In LuaTeX, there is also the case of “bytes”, which behave as character tokens of category code 12 (other) and character code between 1114112 and 1114366. They are used to output individual bytes to files, rather than UTF-8.

Category code 13 (**active**) is discussed below. Input characters can also have several other category codes which do not lead to character tokens for later processing: 0 (**escape**), 5 (**end_line**), 9 (**ignore**), 14 (**comment**), and 15 (**invalid**).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in L^AT_EX3 for most functions and some variables (**tl**, **fp**, **seq**, ...),
- a primitive such as **\def** or **\topmark**, used in L^AT_EX3 for some functions,
- a register such as **\count123**, used in L^AT_EX3 for the implementation of some variables (**int**, **dim**, ...),
- a constant integer such as **\char"56** or **\mathchar"121**,
- a font selection command,
- undefined.

Macros be **\protected** or not, **\long** or not (the opposite of what L^AT_EX3 calls **nopar**), and **\outer** or not (unused in L^AT_EX3). Their **\meaning** takes the form

$\langle properties \rangle \textbf{macro} : \langle parameters \rangle \rightarrow \langle replacement \rangle$

where $\langle properties \rangle$ is among **\protected****\long****\outer**, $\langle parameters \rangle$ describes parameters that the macro expects, such as **#1#2#3**, and $\langle replacement \rangle$ describes how the parameters are manipulated, such as **#2/#1/#3**.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then T_EX scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens “N-type”, as they are suitable to be used as an argument for a function with the signature :N.

Part XVI

The l3prop package

Property lists

L^AT_EX3 implements a “property list” data type, which contain an unordered list of entries each of which consists of a $\langle key \rangle$ and an associated $\langle value \rangle$. The $\langle key \rangle$ and $\langle value \rangle$ may both be any $\langle balanced\ text \rangle$. It is possible to map functions to property lists such that the function is applied to every key–value pair within the list.

Each entry in a property list must have a unique $\langle key \rangle$: if an entry is added to a property list which already contains the $\langle key \rangle$ then the new entry overwrites the existing one. The $\langle keys \rangle$ are compared on a string basis, using the same method as `\str_if_eq:nn`.

Property lists are intended for storing key-based information for use within code. This is in contrast to key–value lists, which are a form of *input* parsed by the `keys` module.

1 Creating and initialising property lists

`\prop_new:N`
`\prop_new:c`

`\prop_new:N` $\langle property\ list \rangle$

Creates a new $\langle property\ list \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle property\ list \rangle$ initially contains no entries.

`\prop_clear:N`
`\prop_clear:c`
`\prop_gclear:N`
`\prop_gclear:c`

`\prop_clear:N` $\langle property\ list \rangle$

Clears all entries from the $\langle property\ list \rangle$.

`\prop_clear_new:N`
`\prop_clear_new:c`
`\prop_gclear_new:N`
`\prop_gclear_new:c`

`\prop_clear_new:N` $\langle property\ list \rangle$

Ensures that the $\langle property\ list \rangle$ exists globally by applying `\prop_new:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

`\prop_set_eq:NN`
`\prop_set_eq:(cN|Nc|cc)`
`\prop_gset_eq:NN`
`\prop_gset_eq:(cN|Nc|cc)`

`\prop_set_eq:NN` $\langle property\ list_1 \rangle$ $\langle property\ list_2 \rangle$

Sets the content of $\langle property\ list_1 \rangle$ equal to that of $\langle property\ list_2 \rangle$.

2 Adding entries to property lists

<code>\prop_put:Nnn</code>	<code>\prop_put:Nnn <property list></code>
<code>\prop_put:(NnV Nno Nnx NVn NVV Non Noo cnn cnV cno cnx cVn cVV con coo)</code>	<code>{<key>} {<value>}</code>
<code>\prop_gput:Nnn</code>	
<code>\prop_gput:(NnV Nno Nnx NVn NVV Non Noo cnn cnV cno cnx cVn cVV con coo)</code>	

Updated: 2012-07-09

Adds an entry to the *<property list>* which may be accessed using the *<key>* and which has *<value>*. Both the *<key>* and *<value>* may contain any *<balanced text>*. The *<key>* is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored. If the *<key>* is already present in the *<property list>*, the existing entry is overwritten by the new *<value>*.

<code>\prop_put_if_new:Nnn</code>	<code>\prop_put_if_new:Nnn <property list> {<key>} {<value>}</code>
<code>\prop_put_if_new:cnn</code>	
<code>\prop_gput_if_new:Nnn</code>	
<code>\prop_gput_if_new:cnn</code>	

If the *<key>* is present in the *<property list>* then no action is taken. If the *<key>* is not present in the *<property list>* then a new entry is added. Both the *<key>* and *<value>* may contain any *<balanced text>*. The *<key>* is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored.

3 Recovering values from property lists

<code>\prop_get:NnN</code>	<code>\prop_get:NnN <property list> {<key>} <tl var></code>
<code>\prop_get:(NVN NoN cnN cVN coN)</code>	

Updated: 2011-08-28

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<token list variable>* is set within the current TeX group. See also `\prop_get:NnNTF`.

<code>\prop_pop:NnN</code>	<code>\prop_pop:NnN <property list> {<key>} <tl var></code>
<code>\prop_pop:(NoN cnN coN)</code>	

Updated: 2011-08-18

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<key>* and *<value>* are then deleted from the property list. Both assignments are local. See also `\prop_pop:NnNTF`.

<code>\prop_gpop:NnN</code>	<code>\prop_gpop:NnN <property list> {<key>} <tl var></code>
<code>\prop_gpop:(NoN cnN coN)</code>	

Updated: 2011-08-18

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<key>* and *<value>* are then deleted from the property list. The *<property list>* is modified globally, while the assignment of the *<token list variable>* is local. See also `\prop_gpop:NnNTF`.

<code>\prop_item:Nn</code> ★	<code>\prop_item:Nn</code> $\langle property list \rangle$ $\{\langle key \rangle\}$
------------------------------	--

<code>\prop_item:cn</code> ★

New: 2014-07-17

Expands to the $\langle value \rangle$ corresponding to the $\langle key \rangle$ in the $\langle property list \rangle$. If the $\langle key \rangle$ is missing, this has an empty expansion.

TeXhackers note: This function is slower than the non-expandable analogue `\prop_get:NnN`. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle value \rangle$ does not expand further when appearing in an `x`-type argument expansion.

4 Modifying property lists

<code>\prop_remove:Nn</code>

<code>\prop_remove:(NV cn cV)</code>

<code>\prop_gremove:Nn</code>

<code>\prop_gremove:(NV cn cV)</code>

New: 2012-05-12

<code>\prop_remove:Nn</code> $\langle property list \rangle$ $\{\langle key \rangle\}$
--

Removes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, *i.e* there is no need to test for the existence of a key before deleting it.

5 Property list conditionals

<code>\prop_if_exist_p:N</code> ★

<code>\prop_if_exist_p:c</code> ★

<code>\prop_if_exist:NTF</code> ★

<code>\prop_if_exist:cTF</code> ★

New: 2012-03-03

<code>\prop_if_exist_p:N</code> $\langle property list \rangle$

<code>\prop_if_exist:NTF</code> $\langle property list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
--

Tests whether the $\langle property list \rangle$ is currently defined. This does not check that the $\langle property list \rangle$ really is a property list variable.

<code>\prop_if_empty_p:N</code> ★

<code>\prop_if_empty_p:c</code> ★

<code>\prop_if_empty:NTF</code> ★

<code>\prop_if_empty:cTF</code> ★

<code>\prop_if_empty_p:N</code> $\langle property list \rangle$

<code>\prop_if_empty:NTF</code> $\langle property list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
--

Tests if the $\langle property list \rangle$ is empty (containing no entries).

<code>\prop_if_in_p:Nn</code>	★	<code>\prop_if_in:NnTF</code> $\langle property list \rangle$ $\{\langle key \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
-------------------------------	---	--

<code>\prop_if_in_p:(NV No cn cV co)</code>	★
---	---

<code>\prop_if_in:NnTF</code>	★
-------------------------------	---

<code>\prop_if_in:(NV No cn cV co)TF</code>	★
---	---

Updated: 2011-09-15

Tests if the $\langle key \rangle$ is present in the $\langle property list \rangle$, making the comparison using the method described by `\str_if_eq:nnTF`.

TeXhackers note: This function iterates through every key–value pair in the $\langle property list \rangle$ and is therefore slower than using the non-expandable `\prop_get:NnNTF`.

6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

<u>\prop_get:NnNTF</u>	<u>\prop_get:NnNTF</u> $\langle \text{property list} \rangle$ $\{\langle \text{key} \rangle\}$ $\langle \text{token list variable} \rangle$
<u>\prop_get:(NVN NoN cnN cVN coN)TF</u>	<u>\prop_get:(NVN NoN cnN cVN coN)TF</u> $\{\langle \text{true code} \rangle\}$ $\{\langle \text{false code} \rangle\}$
Updated: 2012-05-19	

If the $\langle \text{key} \rangle$ is not present in the $\langle \text{property list} \rangle$, leaves the $\langle \text{false code} \rangle$ in the input stream. The value of the $\langle \text{token list variable} \rangle$ is not defined in this case and should not be relied upon. If the $\langle \text{key} \rangle$ is present in the $\langle \text{property list} \rangle$, stores the corresponding $\langle \text{value} \rangle$ in the $\langle \text{token list variable} \rangle$ without removing it from the $\langle \text{property list} \rangle$, then leaves the $\langle \text{true code} \rangle$ in the input stream. The $\langle \text{token list variable} \rangle$ is assigned locally.

<u>\prop_pop:NnNTF</u>	<u>\prop_pop:NnNTF</u> $\langle \text{property list} \rangle$ $\{\langle \text{key} \rangle\}$ $\langle \text{token list variable} \rangle$ $\{\langle \text{true code} \rangle\}$
<u>\prop_pop:cnNTF</u>	<u>\prop_pop:cnNTF</u> $\{\langle \text{false code} \rangle\}$
New: 2011-08-18	If the $\langle \text{key} \rangle$ is not present in the $\langle \text{property list} \rangle$, leaves the $\langle \text{false code} \rangle$ in the input stream. The value of the $\langle \text{token list variable} \rangle$ is not defined in this case and should not be relied upon. If the $\langle \text{key} \rangle$ is present in the $\langle \text{property list} \rangle$, pops the corresponding $\langle \text{value} \rangle$ in the $\langle \text{token list variable} \rangle$, <i>i.e.</i> removes the item from the $\langle \text{property list} \rangle$. Both the $\langle \text{property list} \rangle$ and the $\langle \text{token list variable} \rangle$ are assigned locally.
Updated: 2012-05-19	

<u>\prop_gpop:NnNTF</u>	<u>\prop_gpop:NnNTF</u> $\langle \text{property list} \rangle$ $\{\langle \text{key} \rangle\}$ $\langle \text{token list variable} \rangle$ $\{\langle \text{true code} \rangle\}$
<u>\prop_gpop:cnNTF</u>	<u>\prop_gpop:cnNTF</u> $\{\langle \text{false code} \rangle\}$
New: 2011-08-18	If the $\langle \text{key} \rangle$ is not present in the $\langle \text{property list} \rangle$, leaves the $\langle \text{false code} \rangle$ in the input stream. The value of the $\langle \text{token list variable} \rangle$ is not defined in this case and should not be relied upon. If the $\langle \text{key} \rangle$ is present in the $\langle \text{property list} \rangle$, pops the corresponding $\langle \text{value} \rangle$ in the $\langle \text{token list variable} \rangle$, <i>i.e.</i> removes the item from the $\langle \text{property list} \rangle$. The $\langle \text{property list} \rangle$ is modified globally, while the $\langle \text{token list variable} \rangle$ is assigned locally.
Updated: 2012-05-19	

7 Mapping to property lists

<u>\prop_map_function:NN</u> ☆	<u>\prop_map_function:NN</u> $\langle \text{property list} \rangle$ $\langle \text{function} \rangle$
<u>\prop_map_function:cN</u> ☆	Applies $\langle \text{function} \rangle$ to every $\langle \text{entry} \rangle$ stored in the $\langle \text{property list} \rangle$. The $\langle \text{function} \rangle$ receives two argument for each iteration: the $\langle \text{key} \rangle$ and associated $\langle \text{value} \rangle$. The order in which $\langle \text{entries} \rangle$ are returned is not defined and should not be relied upon.
Updated: 2013-01-28	

<u>\prop_map_inline:Nn</u>	<u>\prop_map_inline:Nn</u> $\langle \text{property list} \rangle$ $\{\langle \text{inline function} \rangle\}$
<u>\prop_map_inline:cn</u>	Applies $\langle \text{inline function} \rangle$ to every $\langle \text{entry} \rangle$ stored within the $\langle \text{property list} \rangle$. The $\langle \text{inline function} \rangle$ should consist of code which receives the $\langle \text{key} \rangle$ as #1 and the $\langle \text{value} \rangle$ as #2. The order in which $\langle \text{entries} \rangle$ are returned is not defined and should not be relied upon.
Updated: 2013-01-08	

\prop_map_break: ☆

Updated: 2012-06-29

\prop_map_break:

Used to terminate a `\prop_map...` function before all entries in the *property list* have been processed. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\prop_map_break:n ☆

Updated: 2012-06-29

\prop_map_break:n {<code>}

Used to terminate a `\prop_map...` function before all entries in the *property list* have been processed, inserting the *code* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \prop_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the *code* is inserted into the input stream. This depends on the design of the mapping function.

8 Viewing property lists

\prop_show:N

\prop_show:c

Updated: 2015-08-01

\prop_show:N *property list*

Displays the entries in the *property list* in the terminal.

<code>\prop_log:N</code>	<code>\prop_log:N</code> \langle <i>property list</i> \rangle
<code>\prop_log:c</code>	Writes the entries in the \langle <i>property list</i> \rangle in the log file.
<hr/> New: 2014-08-12 Updated: 2015-08-01 <hr/>	

9 Scratch property lists

<code>\l_tmpa_prop</code>	Scratch property lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_prop</code>	
<hr/> New: 2012-06-23 <hr/>	

<code>\g_tmpa_prop</code>	Scratch property lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_prop</code>	
<hr/> New: 2012-06-23 <hr/>	

10 Constants

<code>\c_empty_prop</code>	A permanently-empty property list used for internal comparisons.
----------------------------	--

Part XVII

The l3msg package

Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The `l3msg` module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by `l3msg` to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

1 Creating new messages

All messages have to be created before they can be used. The text of messages is automatically wrapped to the length available in the console. As a result, formatting is only needed where it helps to show meaning. In particular, `\\` may be used to force a new line and `_` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided *by one level* using the `/` character. This is used within the message filtering system to allow for example the L^AT_EX kernel messages to belong to the module `LaTeX` while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow to filter out specifically messages from the `submodule`.

```
\msg_new:nnnn
\msg_new:nnn
Updated: 2011-08-16
```

```
\msg_new:nnnn {<module>} {<message>} {<text>} {<more text>}
```

Creates a *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (**#1** to **#4**) can be used: these will be supplied at the time the message is used. An error is raised if the *<message>* already exists.

```
\msg_set:nnnn
\msg_set:nnn
\msg_gset:nnnn
\msg_gset:nnn
```

```
\msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}
```

Sets up the text for a *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (**#1** to **#4**) can be used: these will be supplied at the time the message is used.

<code>\msg_if_exist_p:nn</code> ★	<code>\msg_if_exist_p:nn {<module>} {<message>}</code>
<code>\msg_if_exist:nnTF</code> ★	<code>\msg_if_exist:nnTF {<module>} {<message>} {<true code>} {<false code>}</code>
New: 2012-03-03	Tests whether the <i><message></i> for the <i><module></i> is currently defined.

2 Contextual information for messages

<code>\msg_line_context:</code> ☆	<code>\msg_line_context:</code>
	Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is preceded by the text <code>on line</code> .

<code>\msg_line_number:</code> ★	<code>\msg_line_number:</code>
	Prints the current line number when a message is given.

<code>\msg_fatal_text:n</code> ★	<code>\msg_fatal_text:n {<module>}</code>
	Produces the standard text
	<code>Fatal <module> error</code>
	This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.

<code>\msg_critical_text:n</code> ★	<code>\msg_critical_text:n {<module>}</code>
	Produces the standard text
	<code>Critical <module> error</code>
	This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.

<code>\msg_error_text:n</code> ★	<code>\msg_error_text:n {<module>}</code>
	Produces the standard text
	<code><module> error</code>
	This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.

<code>\msg_warning_text:n</code> ★	<code>\msg_warning_text:n {<module>}</code>
	Produces the standard text
	<code><module> warning</code>
	This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.

`\msg_info_text:n` ★ `\msg_info_text:n {<module>}`

Produces the standard text:

`<module> info`

This function can be redefined to alter the language in which the message is given, using #1 as the name of the `<module>` to be included.

`\msg_see_documentation_text:n` ★ `\msg_see_documentation_text:n {<module>}`

Produces the standard text

`See the <module> documentation for further information.`

This function can be redefined to alter the language in which the message is given, using #1 as the name of the `<module>` to be included.

3 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the x-type variants should be used to expand material.

`\msg_fatal:nnnnnn` `\msg_fatal:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}`
`\msg_fatal:nnxxxx`
`\msg_fatal:nnnnn` Issues `<module>` error `<message>`, passing `<arg one>` to `<arg four>` to the text-creating
`\msg_fatal:nnxxx` functions. After issuing a fatal error the `TEX` run halts.
`\msg_fatal:nnnn`
`\msg_fatal:nnxx`
`\msg_fatal:nnn`
`\msg_fatal:nnx`
`\msg_fatal:nn`

Updated: 2012-08-11

`\msg_critical:nnnnnn` `\msg_critical:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}`
`\msg_critical:nnxxxx`

`\msg_critical:nnnnn` Issues `<module>` error `<message>`, passing `<arg one>` to `<arg four>` to the text-creating
`\msg_critical:nnxxx` functions. After issuing a critical error, `TEX` stops reading the current input file. This
`\msg_critical:nnnn` may halt the `TEX` run (if the current file is the main file) or may abort reading a sub-file.
`\msg_critical:nnxx`

`\msg_critical:nnn` **T_EXhackers note:** The `TEX \endinput` primitive is used to exit the file. In particular,
`\msg_critical:nnx` the rest of the current line remains in the input stream.
`\msg_critical:nn`

Updated: 2012-08-11

<code>\msg_error:nnnnnn</code>	<code>\msg_error:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_error:nnxxxx</code>	
<code>\msg_error:nnnnn</code>	Issues <i><module></i> error <i><message></i> , passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.
<code>\msg_error:nnxxx</code>	
<code>\msg_error:nnnn</code>	
<code>\msg_error:nnxx</code>	
<code>\msg_error:nnn</code>	
<code>\msg_error:nnx</code>	
<code>\msg_error:nn</code>	

Updated: 2012-08-11

<code>\msg_warning:nnnnnn</code>	<code>\msg_warning:nnxxxx {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_warning:nnxxxx</code>	
<code>\msg_warning:nnnnn</code>	Issues <i><module></i> warning <i><message></i> , passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The warning text is added to the log file and the terminal, but the \TeX run is not interrupted.
<code>\msg_warning:nnxxx</code>	
<code>\msg_warning:nnnn</code>	
<code>\msg_warning:nnxx</code>	
<code>\msg_warning:nnn</code>	
<code>\msg_warning:nnx</code>	
<code>\msg_warning:nn</code>	

Updated: 2012-08-11

<code>\msg_info:nnnnnn</code>	<code>\msg_info:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_info:nnxxxx</code>	
<code>\msg_info:nnnnn</code>	Issues <i><module></i> information <i><message></i> , passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The information text is added to the log file.
<code>\msg_info:nnxxx</code>	
<code>\msg_info:nnnn</code>	
<code>\msg_info:nnxx</code>	
<code>\msg_info:nnn</code>	
<code>\msg_info:nnx</code>	
<code>\msg_info:nn</code>	

Updated: 2012-08-11

<code>\msg_log:nnnnnn</code>	<code>\msg_log:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_log:nnxxxx</code>	
<code>\msg_log:nnnnn</code>	Issues <i><module></i> information <i><message></i> , passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The information text is added to the log file: the output is briefer than <code>\msg_info:nnnnnn</code> .
<code>\msg_log:nnxxx</code>	
<code>\msg_log:nnnn</code>	
<code>\msg_log:nnxx</code>	
<code>\msg_log:nnn</code>	
<code>\msg_log:nnx</code>	
<code>\msg_log:nn</code>	

Updated: 2012-08-11

<code>\msg_none:nnnnnn</code>	<code>\msg_none:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_none:nnxxxx</code>	
<code>\msg_none:nnnnn</code>	Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).
<code>\msg_none:nnxxx</code>	
<code>\msg_none:nnnn</code>	
<code>\msg_none:nnxx</code>	
<code>\msg_none:nnn</code>	
<code>\msg_none:nnx</code>	
<code>\msg_none:nn</code>	

Updated: 2012-08-11

4 Redirecting messages

Each message has a “name”, which can be used to alter the behaviour of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some-text } { Some-more-text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this raises an error. However, we could alter the behaviour with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class raises an error immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as $A \rightarrow B$, $B \rightarrow C$ and $C \rightarrow A$ in this order, then the $A \rightarrow B$ redirection is cancelled.

<code>\msg_redirect_class:nn</code>	<code>\msg_redirect_class:nn {<class one>} {<class two>}</code>
-------------------------------------	---

Updated: 2012-04-27

Changes the behaviour of messages of *<class one>* so that they are processed using the code for those of *<class two>*.

`\msg_log:n`

`\msg_log:n {<text>}`

New: 2012-06-28

Writes to the log file with the $\langle text \rangle$ laid out in the format

```
.....  
. <text>  
.....
```

where the $\langle text \rangle$ is wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

`\msg_term:n`

`\msg_term:n {<text>}`

New: 2012-06-28

Writes to the terminal and log file with the $\langle text \rangle$ laid out in the format

```
*****  
* <text>  
*****
```

where the $\langle text \rangle$ is wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

Part XVIII

The l3file package

File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files T_EX attempts to locate them using both the operating system path and entries in the T_EX file database (most T_EX systems use such a database). Thus the “current path” for T_EX is somewhat broader than that for other programs.

For functions which expect a *<file name>* argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in `\l_char_active_seq`) are *not* expanded, allowing the direct use of these in file names. File names are quoted using `"` tokens if they contain spaces: as a result, `"` tokens are *not* permitted in file names.

1 Input–output stream management

As T_EX engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in L^AT_EX3. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

<code>\ior_new:N</code>	<code>\ior_new:N <stream></code>
<code>\ior_new:c</code>	<code>\ior_new:c <stream></code>
<code>\iow_new:N</code>	
<code>\iow_new:c</code>	
<hr/> New: 2011-09-26 Updated: 2011-12-27	
<code>\ior_open:Nn</code>	<code>\ior_open:Nn <stream> {<file name>}</code>
<code>\ior_open:cn</code>	
<hr/> Updated: 2012-02-10	
	Globally reserves the name of the <i><stream></i> , either for reading or for writing as appropriate. The <i><stream></i> is not opened until the appropriate <code>\..._open:Nn</code> function is used. Attempting to use a <i><stream></i> which has not been opened is an error, and the <i><stream></i> will behave as the corresponding <code>\c_term_....</code>
	Opens <i><file name></i> for reading using <i><stream></i> as the control sequence for file access. If the <i><stream></i> was already open it is closed before the new operation begins. The <i><stream></i> is available for access immediately and will remain allocated to <i><file name></i> until a <code>\ior_close:N</code> instruction is given or the T _E X run ends. If the file is not found, an error is raised.

<hr/> <code>\ior_open:NnTF</code> <hr/>	<code>\ior_open:NnTF <stream> {<file name>} {<true code>} {<false code>}</code>
<code>\ior_open:cnTF</code> <hr/>	
<code>New: 2013-01-12</code> <hr/>	Opens <i><file name></i> for reading using <i><stream></i> as the control sequence for file access. If the <i><stream></i> was already open it is closed before the new operation begins. The <i><stream></i> is available for access immediately and will remain allocated to <i><file name></i> until a <code>\ior_close:N</code> instruction is given or the T _E X run ends. The <i><true code></i> is then inserted into the input stream. If the file is not found, no error is raised and the <i><false code></i> is inserted into the input stream.
<hr/>	
<code>\iow_open:Nn</code> <hr/>	<code>\iow_open:Nn <stream> {<file name>}</code>
<code>\iow_open:cn</code> <hr/>	
<code>Updated: 2012-02-09</code> <hr/>	Opens <i><file name></i> for writing using <i><stream></i> as the control sequence for file access. If the <i><stream></i> was already open it is closed before the new operation begins. The <i><stream></i> is available for access immediately and will remain allocated to <i><file name></i> until a <code>\iow_close:N</code> instruction is given or the T _E X run ends. Opening a file for writing clears any existing content in the file (<i>i.e.</i> writing is <i>not</i> additive).
<hr/>	
<code>\ior_close:N</code> <hr/>	<code>\ior_close:N <stream></code>
<code>\ior_close:c</code> <hr/>	<code>\iow_close:N <stream></code>
<code>\iow_close:N</code> <hr/>	
<code>\iow_close:c</code> <hr/>	Closes the <i><stream></i> . Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.
<code>Updated: 2012-07-31</code> <hr/>	
<hr/>	
<code>\ior_show_list:</code> <hr/>	<code>\ior_show_list:</code>
<code>\ior_log_list:</code> <hr/>	<code>\ior_log_list:</code>
<code>\iow_show_list:</code> <hr/>	<code>\iow_show_list:</code>
<code>\iow_log_list:</code> <hr/>	<code>\iow_log_list:</code>
<code>New: 2017-06-27</code> <hr/>	Display (to the terminal or log file) a list of the file names associated with each open (read or write) stream. This is intended for tracking down problems.

1.1 Reading from files

<code>\ior_get:NN</code>	<code>\ior_get:NN <stream> <token list variable></code>
--------------------------	---

New: 2012-06-24

Function that reads one or more lines (until an equal number of left and right braces are found) from the input *<stream>* and stores the result locally in the *<token list>* variable. If the *<stream>* is not open, input is requested from the terminal. The material read from the *<stream>* is tokenized by T_EX according to the category codes and `\endlinechar` in force when the function is used. Assuming normal settings, any lines which do not end in a comment character % have the line ending converted to a space, so for example input

```
a b c
```

results in a token list `a_b_c_`. Any blank line is converted to the token `\par`. Therefore, blank lines can be skipped by using a test such as

```
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain `\par` tokens.

T_EXhackers note: This protected macro is a wrapper around the T_EX primitive `\read`. Regardless of settings, T_EX replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code `\endlinechar`, omitted if `\endlinechar` is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

<code>\ior_str_get:NN</code>	<code>\ior_str_get:NN <stream> <token list variable></code>
------------------------------	---

New: 2016-12-04

Function that reads one line from the input *<stream>* and stores the result locally in the *<token list>* variable. If the *<stream>* is not open, input is requested from the terminal. The material is read from the *<stream>* as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the *<token list variable>* being empty. Unlike `\ior_get:NN`, line ends do not receive any special treatment. Thus input

```
a b c
```

results in a token list `a b c` with the letters `a`, `b`, and `c` having category code 12.

T_EXhackers note: This protected macro is a wrapper around the ϵ -T_EX primitive `\readline`. Regardless of settings, T_EX removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of `\ior_str_get:NN`.

<hr/> <code>\ior_map_inline:Nn</code> <hr/>	<code>\ior_map_inline:Nn <stream> {<inline function>}</code>
<hr/> New: 2012-02-11 <hr/>	Applies the <i><inline function></i> to each set of <i><lines></i> obtained by calling <code>\ior_get:NN</code> until reaching the end of the file. T _E X ignores any trailing new-line marker from the file it reads. The <i><inline function></i> should consist of code which receives the <i><line></i> as #1.
<hr/> <code>\ior_str_map_inline:Nn</code> <hr/>	<code>\ior_str_map_inline:Nn <stream> {<inline function>}</code>
<hr/> New: 2012-02-11 <hr/>	Applies the <i><inline function></i> to every <i><line></i> in the <i><stream></i> . The material is read from the <i><stream></i> as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The <i><inline function></i> should consist of code which receives the <i><line></i> as #1. Note that T _E X removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T _E X also ignores any trailing new-line marker from the file it reads.
<hr/> <code>\ior_map_break:</code> <hr/>	<code>\ior_map_break:</code>
<hr/> New: 2012-06-29 <hr/>	Used to terminate a <code>\ior_map...</code> function before all lines from the <i><stream></i> have been processed. This normally takes place within a conditional statement, for example <pre> \ior_map_inline:Nn \l_my_ior { \str_if_eq:nnTF { #1 } { bingo } { \ior_map_break: } { % Do something useful } } </pre> <p>Use outside of a <code>\ior_map...</code> scenario leads to low level T_EX errors.</p> <p>T_EXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.</p>

\ior_map_break:n

New: 2012-06-29

\ior_map_break:n {<code>}

Used to terminate a **\ior_map_...** function before all lines in the <stream> have been processed, inserting the <code> after the mapping has ended. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
{
  \str_if_eq:nnTF { #1 } { bingo }
  { \ior_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a **\ior_map_...** scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the <code> is inserted into the input stream. This depends on the design of the mapping function.

\ior_if_eof_p:N ★**\ior_if_eof:NTF** ★

Updated: 2012-02-10

\ior_if_eof_p:N <stream>**\ior_if_eof:NTF** <stream> {<true code>} {<false code>}

Tests if the end of a <stream> has been reached during a reading operation. The test also returns a **true** value if the <stream> is not open.

2 Writing to files

\iow_now:Nn**\iow_now:(Nx|cn|cx)**

Updated: 2012-06-05

\iow_now:Nn <stream> {<tokens>}

This functions writes <tokens> to the specified <stream> immediately (*i.e.* the write operation is called on expansion of **\iow_now:Nn**).

\iow_log:n**\iow_log:x****\iow_log:n** {<tokens>}

This function writes the given <tokens> to the log (transcript) file immediately: it is a dedicated version of **\iow_now:Nn**.

\iow_term:n**\iow_term:x****\iow_term:n** {<tokens>}

This function writes the given <tokens> to the terminal file immediately: it is a dedicated version of **\iow_now:Nn**.

<hr/> <code>\iow_shipout:Nn</code> <code>\iow_shipout:(Nx cn cx)</code> <hr/>	<code>\iow_shipout:Nn <stream> {<tokens>}</code> This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (<i>i.e.</i> at shipout). The x -type variants expand the $\langle tokens \rangle$ at the point where the function is used but <i>not</i> when the resulting tokens are written to the $\langle stream \rangle$ (<i>cf.</i> <code>\iow_shipout_x:Nn</code>). <p>T_EXhackers note: When using <code>expl3</code> with a format other than L^AT_EX, new line characters inserted using <code>\iow_newline:</code> or using the line-wrapping code <code>\iow_wrap:nnnN</code> are not recognized in the argument of <code>\iow_shipout:Nn</code>. This may lead to the insertion of additional unwanted line-breaks.</p>
<hr/> <code>\iow_shipout_x:Nn</code> <code>\iow_shipout_x:(Nx cn cx)</code> <hr/> <div>Updated: 2012-09-08</div> <hr/>	<code>\iow_shipout_x:Nn <stream> {<tokens>}</code> This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (<i>i.e.</i> at shipout). The $\langle tokens \rangle$ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer). <p>T_EXhackers note: This is a wrapper around the T_EX primitive <code>\write</code>. When using <code>expl3</code> with a format other than L^AT_EX, new line characters inserted using <code>\iow_newline:</code> or using the line-wrapping code <code>\iow_wrap:nnnN</code> are not recognized in the argument of <code>\iow_shipout:Nn</code>. This may lead to the insertion of additional unwanted line-breaks.</p>
<hr/> <code>\iow_char:N</code> ★ <hr/>	<code>\iow_char:N \<char></code> Inserts $\langle char \rangle$ into the output stream. Useful when trying to write difficult characters such as %, {, }, <i>etc.</i> in messages, for example: <code>\iow_now:Nx \g_my_iow { \iow_char:N \{ text \iow_char:N \} }</code> The function has no effect if writing is taking place without expansion (<i>e.g.</i> in the second argument of <code>\iow_now:Nn</code>).
<hr/> <code>\iow_newline:</code> ★ <hr/>	<code>\iow_newline:</code> Function to add a new line within the $\langle tokens \rangle$ written to a file. The function has no effect if writing is taking place without expansion (<i>e.g.</i> in the second argument of <code>\iow_now:Nn</code>). <p>T_EXhackers note: When using <code>expl3</code> with a format other than L^AT_EX, the character inserted by <code>\iow_newline:</code> is not recognized by T_EX, which may lead to the insertion of additional unwanted line-breaks. This issue only affects <code>\iow_shipout:Nn</code>, <code>\iow_shipout_x:Nn</code> and direct uses of primitive operations.</p>

2.1 Wrapping lines in output

`\iow_wrap:nnnN`

New: 2012-06-28
Updated: 2017-12-04

`\iow_wrap:nnnN` $\langle text \rangle$ $\langle run-on\ text \rangle$ $\langle set\ up \rangle$ $\langle function \rangle$

This function wraps the $\langle text \rangle$ to a fixed number of characters per line. At the start of each line which is wrapped, the $\langle run-on\ text \rangle$ is inserted. The line character count targeted is the value of `\l_iow_line_count_int` minus the number of characters in the $\langle run-on\ text \rangle$ for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The $\langle text \rangle$ and $\langle run-on\ text \rangle$ are exhaustively expanded by the function, with the following substitutions:

- `\` or `\iow_newline`: may be used to force a new line,
- `_` may be used to represent a forced space (for example after a control sequence),
- `\#`, `\%`, `\{`, `\}`, `\~` may be used to represent the corresponding character,
- `\iow_indent:n` may be used to indent a part of the $\langle text \rangle$ (not the $\langle run-on\ text \rangle$).

Additional functions may be added to the wrapping by using the $\langle set\ up \rangle$, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the $\langle text \rangle$ which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, *etc.*

The result of the wrapping operation is passed as a braced argument to the $\langle function \rangle$, which is typically a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the $\langle function \rangle$) consists of characters of category “other” (category code 12), with the exception of spaces which have category “space” (category code 10). This means that the output does *not* expand further when written to a file.

T_EXhackers note: Internally, `\iow_wrap:nnnN` carries out an `x`-type expansion on the $\langle text \rangle$ to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` *could* be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the $\langle text \rangle$.

`\iow_indent:n`

New: 2011-09-21

`\iow_indent:n` $\langle text \rangle$

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents $\langle text \rangle$ by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the $\langle text \rangle$. In case the indented $\langle text \rangle$ should appear on separate lines from the surrounding text, use `\` to force line breaks.

`\l_iow_line_count_int`

New: 2012-06-24

The maximum number of characters in a line to be written by the `\iow_wrap:nnnN` function. This value depends on the T_EX system in use: the standard value is 78, which is typically correct for unmodified T_EXlive and MiK_TE_X systems.

2.2 Constant input–output streams, and variables

<code>\c_term_ior</code>	Constant input stream for reading from the terminal. Reading from this stream using <code>\ior_get:NN</code> or similar results in a prompt from T _E X of the form
--------------------------	---

`<tl>=`

<code>\g_tmpa_ior</code> <code>\g_tmpb_ior</code>	Scratch input stream for global use. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	--

New: 2017-12-11

<code>\c_log_ior</code> <code>\c_term_ior</code>	Constant output streams for writing to the log and to the terminal (plus the log), respectively.
---	--

<code>\g_tmpa_ior</code> <code>\g_tmpb_ior</code>	Scratch output stream for global use. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	---

New: 2017-12-11

2.3 Primitive conditionals

<code>\if_eof:w</code> ★	<pre> \if_eof:w <stream> <true code> \else: <false code> \fi: </pre> <p>Tests if the <code><stream></code> returns “end of file”, which is true for non-existent files. The <code>\else:</code> branch is optional.</p>
--------------------------	---

T_EXhackers note: This is the T_EX primitive `\ifeof`.

3 File operation functions

<code>\g_file_curr_dir_str</code> <code>\g_file_curr_name_str</code> <code>\g_file_curr_ext_str</code>	Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (<i>i.e.</i> if it is in the T _E X search path), and does <i>not</i> end in / other than the case that it is exactly equal to the root directory. The <code><name></code> and <code><ext></code> parts together make up the file name, thus the <code><name></code> part may be thought of as the “job name” for the current file. Note that T _E X does not provide information on the <code><ext></code> part for the main (top level) file and that this file always has an empty <code><dir></code> component. Also, the <code><name></code> here will be equal to <code>\c_sys_jobname_str</code> , which may be different from the real file name (if set using <code>--jobname</code> , for example).
--	--

New: 2017-06-21

`\l_file_search_path_seq`

New: 2017-06-18

Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and should not include the trailing slash. The entries are not expanded when used so may contain active characters but should not feature any variable content. Spaces need not be quoted.

T_EXhackers note: When working as a package in L^AT_EX 2_ε, `expl3` will automatically append the current `\input@path` to the set of values from `\l_file_search_path_seq`.

`\file_if_exist:nTF`

Updated: 2012-02-10

`\file_if_exist:nTF {<file name>} {<true code>} {<false code>}`

Searches for `<file name>` using the current T_EX search path and the additional paths controlled by `\l_file_search_path_seq`.

`\file_get_full_name:nN`
`\file_get_full_name:VN`

Updated: 2017-06-26

`\file_get_full_name:nN {<file name>} <str var>`

Searches for `<file name>` in the path as detailed for `\file_if_exist:nTF`, and if found sets the `<str var>` the fully-qualified name of the file, *i.e.* the path and file name. This includes an extension `.tex` when the given `<file name>` has no extension but the file found has that extension. If the file is not found then the `<str var>` is empty.

`\file_parse_full_name:nNNN`

New: 2017-06-23
Updated: 2017-06-26

`\file_parse_full_name:nNNN {<full name>} <dir> <name> <ext>`

Parses the `<full name>` and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The `<dir>`: everything up to the last / (path separator) in the `<file path>`. As with system PATH variables and related functions, the `<dir>` does *not* include the trailing / unless it points to the root directory. If there is no path (only a file name), `<dir>` is empty.
- The `<name>`: everything after the last / up to the last ., where both of those characters are optional. The `<name>` may contain multiple . characters. It is empty if `<full name>` consists only of a directory name.
- The `<ext>`: everything after the last . (including the dot). The `<ext>` is empty if there is no . after the last /.

This function does not expand the `<full name>` before turning it to a string. It assume that the `<full name>` either contains no quote (") characters or is surrounded by a pair of quotes.

`\file_input:n`

Updated: 2017-06-26

`\file_input:n {<file name>}`

Searches for `<file name>` in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional L^AT_EX source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.

`\file_show_list:`
`\file_log_list:`

`\file_show_list:`
`\file_log_list:`

These functions list all files loaded by L^AT_EX 2_ε commands that populate `\@filelist` or by `\file_input:n`. While `\file_show_list:` displays the list in the terminal, `\file_log_list:` outputs it to the log file only.

Part XIX

The l3skip package

Dimensions and skips

L^AT_EX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in μ). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

1 Creating and initialising `dim` variables

<code>\dim_new:N</code>
<code>\dim_new:c</code>

`\dim_new:N` $\langle dimension \rangle$

Creates a new $\langle dimension \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle dimension \rangle$ is initially equal to 0pt.

<code>\dim_const:Nn</code>
<code>\dim_const:cn</code>

`\dim_const:Nn` $\langle dimension \rangle$ $\{ \langle dimension expression \rangle \}$

Creates a new constant $\langle dimension \rangle$ or raises an error if the name is already taken. The value of the $\langle dimension \rangle$ is set globally to the $\langle dimension expression \rangle$.

New: 2012-03-05

<code>\dim_zero:N</code>
<code>\dim_zero:c</code>
<code>\dim_gzero:N</code>
<code>\dim_gzero:c</code>

`\dim_zero:N` $\langle dimension \rangle$

Sets $\langle dimension \rangle$ to 0pt.

<code>\dim_zero_new:N</code>
<code>\dim_zero_new:c</code>
<code>\dim_gzero_new:N</code>
<code>\dim_gzero_new:c</code>

`\dim_zero_new:N` $\langle dimension \rangle$

Ensures that the $\langle dimension \rangle$ exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the $\langle dimension \rangle$ set to zero.

New: 2012-01-07

<code>\dim_if_exist_p:N</code> ★
<code>\dim_if_exist_p:c</code> ★
<code>\dim_if_exist:N\underline{TF}</code> ★
<code>\dim_if_exist:c\underline{TF}</code> ★

`\dim_if_exist_p:N` $\langle dimension \rangle$

`\dim_if_exist:N \underline{TF}` $\langle dimension \rangle$ $\{ \langle true code \rangle \}$ $\{ \langle false code \rangle \}$

Tests whether the $\langle dimension \rangle$ is currently defined. This does not check that the $\langle dimension \rangle$ really is a dimension variable.

New: 2012-03-03

2 Setting dim variables

<code>\dim_add:Nn</code>	<code>\dim_add:Nn <dimension> {<dimension expression>}</code>
<code>\dim_add:cn</code>	
<code>\dim_gadd:Nn</code>	Adds the result of the $\langle dimension\ expression \rangle$ to the current content of the $\langle dimension \rangle$.
<code>\dim_gadd:cn</code>	

Updated: 2011-10-22

<code>\dim_set:Nn</code>	<code>\dim_set:Nn <dimension> {<dimension expression>}</code>
<code>\dim_set:cn</code>	
<code>\dim_gset:Nn</code>	Sets $\langle dimension \rangle$ to the value of $\langle dimension\ expression \rangle$, which must evaluate to a length with units.
<code>\dim_gset:cn</code>	

Updated: 2011-10-22

<code>\dim_set_eq:NN</code>	<code>\dim_set_eq:NN <dimension₁> <dimension₂></code>
<code>\dim_set_eq:(cN Nc cc)</code>	
<code>\dim_gset_eq:NN</code>	Sets the content of $\langle dimension_1 \rangle$ equal to that of $\langle dimension_2 \rangle$.
<code>\dim_gset_eq:(cN Nc cc)</code>	

<code>\dim_sub:Nn</code>	<code>\dim_sub:Nn <dimension> {<dimension expression>}</code>
<code>\dim_sub:cn</code>	
<code>\dim_gsub:Nn</code>	Subtracts the result of the $\langle dimension\ expression \rangle$ from the current content of the $\langle dimension \rangle$.
<code>\dim_gsub:cn</code>	

Updated: 2011-10-22

3 Utilities for dimension calculations

<code>\dim_abs:n</code>	★ <code>\dim_abs:n {<dimexpr>}</code>
Updated: 2012-09-26	Converts the $\langle dimexpr \rangle$ to its absolute value, leaving the result in the input stream as a $\langle dimension\ denotation \rangle$.

<code>\dim_max:nn</code>	★ <code>\dim_max:nn {<dimexpr₁>} {<dimexpr₂>}</code>
<code>\dim_min:nn</code>	★ <code>\dim_min:nn {<dimexpr₁>} {<dimexpr₂>}</code>
New: 2012-09-09	
Updated: 2012-09-26	Evaluates the two $\langle dimension\ expressions \rangle$ and leaves either the maximum or minimum value in the input stream as appropriate, as a $\langle dimension\ denotation \rangle$.

`\dim_ratio:nn` ☆

Updated: 2011-10-22

`\dim_ratio:nn` { $\langle dimexpr_1 \rangle$ } { $\langle dimexpr_2 \rangle$ }

Parses the two $\langle dimension expressions \rangle$ and converts the ratio of the two to a form suitable for use inside a $\langle dimension expression \rangle$. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
{ 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of `\dim_ratio:nn` on full expansion is a ration expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Nx \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

displays 327680/655360 on the terminal.

4 Dimension expression conditionals

`\dim_compare_p:nNn` ☆

`\dim_compare:nNnTF` ☆

`\dim_compare_p:nNn` { $\langle dimexpr_1 \rangle$ } $\langle relation \rangle$ { $\langle dimexpr_2 \rangle$ }

`\dim_compare:nNnTF`
{ $\langle dimexpr_1 \rangle$ } $\langle relation \rangle$ { $\langle dimexpr_2 \rangle$ }
{ $\langle true code \rangle$ } { $\langle false code \rangle$ }

This function first evaluates each of the $\langle dimension expressions \rangle$ as described for `\dim_eval:n`. The two results are then compared using the $\langle relation \rangle$:

Equal	=
Greater than	>
Less than	<

<code>\dim_compare_p:n</code> ★ <code>\dim_compare:nTF</code> ★	<code>\dim_compare_p:n</code> { $\langle dimexpr_1 \rangle$ $\langle relation_1 \rangle$... $\langle dimexpr_N \rangle$ $\langle relation_N \rangle$ $\langle dimexpr_{N+1} \rangle$ } <code>\dim_compare:nTF</code> { $\langle dimexpr_1 \rangle$ $\langle relation_1 \rangle$... $\langle dimexpr_N \rangle$ $\langle relation_N \rangle$ $\langle dimexpr_{N+1} \rangle$ } { $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }
--	--

Updated: 2013-01-13

This function evaluates the $\langle dimension\ expressions \rangle$ as described for `\dim_eval:n` and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle dimexpr_1 \rangle$ and $\langle dimexpr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle dimexpr_2 \rangle$ and $\langle dimexpr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle dimexpr_N \rangle$ and $\langle dimexpr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields `true` if all comparisons are `true`. Each $\langle dimension\ expression \rangle$ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other $\langle dimension\ expression \rangle$ is evaluated and no other comparison is performed. The $\langle relations \rangle$ can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

<code>\dim_case:nn</code> ★	<code>\dim_case:nnTF {⟨test dimension expression⟩}</code>
<code>\dim_case:nnTF</code> ★	<code>{</code> <code> {⟨dimexpr case₁⟩} {⟨code case₁⟩}</code> <code> {⟨dimexpr case₂⟩} {⟨code case₂⟩}</code> <code> ...</code> <code> {⟨dimexpr case_n⟩} {⟨code case_n⟩}</code> <code>}</code> <code>{⟨true code⟩}</code> <code>{⟨false code⟩}</code>

New: 2013-07-24

This function evaluates the *⟨test dimension expression⟩* and compares this in turn to each of the *⟨dimension expression cases⟩*. If the two are equal then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\dim_case:nn`, which does nothing if there is no match, is also available. For example

```

\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
{ 2 \l_tmpa_dim }
{
  { 5 pt }      { Small }
  { 4 pt + 6 pt } { Medium }
  { - 10 pt }   { Negative }
}
{ No idea! }
```

leaves “Medium” in the input stream.

5 Dimension expression loops

<code>\dim_do_until:nNnn</code> ☆	<code>\dim_do_until:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`. If the test is **false** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **true**.

<code>\dim_do_while:nNnn</code> ☆	<code>\dim_do_while:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Places the *⟨code⟩* in the input stream for T_EX to process, and then evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`. If the test is **true** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **false**.

<code>\dim_until_do:nNnn</code> ☆	<code>\dim_until_do:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}</code>
-----------------------------------	---

Evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`, and then places the *⟨code⟩* in the input stream if the *⟨relation⟩* is **false**. After the *⟨code⟩* has been processed by T_EX the test is repeated, and a loop occurs until the test is **true**.

<hr/> <code>\dim_while_do:nNnn</code> ☆ <hr/>	<code>\dim_while_do:nNnn {<dimexpr1>} <relation> {<dimexpr2>} {<code>}</code>
	Evaluates the relationship between the two <i><dimension expressions></i> as described for <code>\dim_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .
<hr/> <code>\dim_do_until:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_do_until:nn {<dimension relation>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is false then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is true .
<hr/> <code>\dim_do_while:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_do_while:nn {<dimension relation>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is true then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is false .
<hr/> <code>\dim_until_do:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_until_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\dim_while_do:nn</code> ☆ <hr/> <div>Updated: 2013-01-13</div>	<code>\dim_while_do:nn {<dimension relation>} {<code>}</code>
	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .

6 Dimension step functions

<hr/> <code>\dim_step_function:nnnN</code> ☆ <hr/> <div>New: 2018-02-18</div>	<code>\dim_step_function:nnnN {<initial value>} {<step>} {<final value>} <function></code>
	This function first evaluates the <i><initial value></i> , <i><step></i> and <i><final value></i> , all of which should be dimension expressions. The <i><function></i> is then placed in front of each <i><value></i> from the <i><initial value></i> to the <i><final value></i> in turn (using <i><step></i> between each <i><value></i>). The <i><step></i> must be non-zero. If the <i><step></i> is positive, the loop stops when the <i><value></i> becomes larger than the <i><final value></i> . If the <i><step></i> is negative, the loop stops when the <i><value></i> becomes smaller than the <i><final value></i> . The <i><function></i> should absorb one argument.
<hr/> <code>\dim_step_inline:nnnn</code> <hr/> <div>New: 2018-02-18</div>	<code>\dim_step_inline:nnnn {<initial value>} {<step>} {<final value>} {<code>}</code>
	This function first evaluates the <i><initial value></i> , <i><step></i> and <i><final value></i> , all of which should be dimension expressions. Then for each <i><value></i> from the <i><initial value></i> to the <i><final value></i> in turn (using <i><step></i> between each <i><value></i>), the <i><code></i> is inserted into the input stream with #1 replaced by the current <i><value></i> . Thus the <i><code></i> should define a function of one argument (#1).

`\dim_step_variable:nnnNn`
 New: 2018-02-18

`\dim_step_variable:nnnNn`
`{\langle initial value \rangle}{\langle step \rangle}{\langle final value \rangle}{\langle tl var \rangle}{\langle code \rangle}`

This function first evaluates the $\langle initial value \rangle$, $\langle step \rangle$ and $\langle final value \rangle$, all of which should be dimension expressions. Then for each $\langle value \rangle$ from the $\langle initial value \rangle$ to the $\langle final value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$), the $\langle code \rangle$ is inserted into the input stream, with the $\langle tl var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl var \rangle$.

7 Using dim expressions and variables

`\dim_eval:n` ★
 Updated: 2011-10-22

`\dim_eval:n {\langle dimension expression \rangle}`

Evaluates the $\langle dimension expression \rangle$, expanding any dimensions and token list variables within the $\langle expression \rangle$ to their content (without requiring `\dim_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle dimension denotation \rangle$ after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a \TeX -style assignment as it is *not* an $\langle internal dimension \rangle$.

`\dim_use:N` ★
`\dim_use:c` ★

`\dim_use:N \langle dimension \rangle`

Recovers the content of a $\langle dimension \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of `\dim_eval:n`).

\TeX hackers note: `\dim_use:N` is the \TeX primitive `\the`: this is one of several \LaTeX 3 names for this primitive.

`\dim_to_decimal:n` ★
 New: 2014-07-15

`\dim_to_decimal:n {\langle dimexpr \rangle}`

Evaluates the $\langle dimension expression \rangle$, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by \TeX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

`\dim_to_decimal:n { 1bp }`

leaves 1.00374 in the input stream, *i.e.* the magnitude of one “big point” when converted to (\TeX) points.

<code>\dim_to_decimal_in_bp:n</code> ★	<code>\dim_to_decimal_in_bp:n {⟨dimexpr⟩}</code>
New: 2014-07-15	Evaluates the <i>⟨dimension expression⟩</i> , and leaves the result, expressed in big points (bp) in the input stream, with <i>no units</i> . The result is rounded by T _E X to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_bp:n { 1pt }
```

leaves 0.99628 in the input stream, *i.e.* the magnitude of one (T_EX) point when converted to big points.

<code>\dim_to_decimal_in_sp:n</code> ★	<code>\dim_to_decimal_in_sp:n {⟨dimexpr⟩}</code>
New: 2015-05-18	Evaluates the <i>⟨dimension expression⟩</i> , and leaves the result, expressed in scaled points (sp) in the input stream, with <i>no units</i> . The result is necessarily an integer.

<code>\dim_to_decimal_in_unit:nn</code> ★	<code>\dim_to_decimal_in_unit:nn {⟨dimexpr₁⟩} {⟨dimexpr₂⟩}</code>
New: 2014-07-15	

Evaluates the *⟨dimension expressions⟩*, and leaves the value of *⟨dimexpr₁⟩*, expressed in a unit given by *⟨dimexpr₂⟩*, in the input stream. The result is a decimal number, rounded by T_EX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_unit:nn { 1bp } { 1mm }
```

leaves 0.35277 in the input stream, *i.e.* the magnitude of one big point when converted to millimetres.

Note that this function is not optimised for any particular output and as such may give different results to `\dim_to_decimal_in_bp:n` or `\dim_to_decimal_in_sp:n`. In particular, the latter is able to take a wider range of input values as it is not limited by the ability to calculate a ratio using ε -T_EX primitives, which is required internally by `\dim_to_decimal_in_unit:nn`.

<code>\dim_to_fp:n</code> ★	<code>\dim_to_fp:n {⟨dimexpr⟩}</code>
New: 2012-05-08	Expands to an internal floating point number equal to the value of the <i>⟨dimexpr⟩</i> in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, <code>\dim_to_fp:n</code> can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

8 Viewing dim variables

<code>\dim_show:N</code>	<code>\dim_show:N ⟨dimension⟩</code>
<code>\dim_show:c</code>	Displays the value of the <i>⟨dimension⟩</i> on the terminal.

<hr/> <code>\dim_show:n</code> <hr/>	<code>\dim_show:n {⟨dimension expression⟩}</code>
New: 2011-11-22 Updated: 2015-08-07	Displays the result of evaluating the $\langle dimension\ expression \rangle$ on the terminal.
<hr/> <code>\dim_log:N</code> <code>\dim_log:c</code> <hr/>	<code>\dim_log:N ⟨dimension⟩</code>
New: 2014-08-22 Updated: 2015-08-03	Writes the value of the $\langle dimension \rangle$ in the log file.
<hr/> <code>\dim_log:n</code> <hr/>	<code>\dim_log:n {⟨dimension expression⟩}</code>
New: 2014-08-22 Updated: 2015-08-07	Writes the result of evaluating the $\langle dimension\ expression \rangle$ in the log file.

9 Constant dimensions

<hr/> <code>\c_max_dim</code> <hr/>	The maximum value that can be stored as a dimension. This can also be used as a component of a skip.
<hr/> <code>\c_zero_dim</code> <hr/>	A zero length as a dimension. This can also be used as a component of a skip.

10 Scratch dimensions

<hr/> <code>\l_tmpa_dim</code> <code>\l_tmpb_dim</code> <hr/>	Scratch dimension for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_dim</code> <code>\g_tmpb_dim</code> <hr/>	Scratch dimension for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

11 Creating and initialising skip variables

<hr/> <code>\skip_new:N</code> <code>\skip_new:c</code> <hr/>	<code>\skip_new:N ⟨skip⟩</code>
	Creates a new $\langle skip \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle skip \rangle$ is initially equal to 0 pt.
<hr/> <code>\skip_const:Nn</code> <code>\skip_const:cn</code> <hr/>	<code>\skip_const:Nn ⟨skip⟩ {⟨skip expression⟩}</code>
New: 2012-03-05	Creates a new constant $\langle skip \rangle$ or raises an error if the name is already taken. The value of the $\langle skip \rangle$ is set globally to the $\langle skip\ expression \rangle$.

<code>\skip_zero:N</code>	<code>\skip_zero:N</code> $\langle skip \rangle$
<code>\skip_zero:c</code>	Sets $\langle skip \rangle$ to 0 pt.
<code>\skip_gzero:N</code>	
<code>\skip_gzero:c</code>	

<code>\skip_zero_new:N</code>	<code>\skip_zero_new:N</code> $\langle skip \rangle$
<code>\skip_zero_new:c</code>	Ensures that the $\langle skip \rangle$ exists globally by applying <code>\skip_new:N</code> if necessary, then applies
<code>\skip_gzero_new:N</code>	<code>\skip_(g)zero:N</code> to leave the $\langle skip \rangle$ set to zero.
<code>\skip_gzero_new:c</code>	

New: 2012-01-07

<code>\skip_if_exist_p:N</code> ★	<code>\skip_if_exist_p:N</code> $\langle skip \rangle$
<code>\skip_if_exist_p:c</code> ★	<code>\skip_if_exist:NTF</code> $\langle skip \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\skip_if_exist:NTF</code> ★	Tests whether the $\langle skip \rangle$ is currently defined. This does not check that the $\langle skip \rangle$ really
<code>\skip_if_exist:cTF</code> ★	is a skip variable.

New: 2012-03-03

12 Setting skip variables

<code>\skip_add:Nn</code>	<code>\skip_add:Nn</code> $\langle skip \rangle$ $\{\langle skip\ expression \rangle\}$
<code>\skip_add:cn</code>	Adds the result of the $\langle skip\ expression \rangle$ to the current content of the $\langle skip \rangle$.
<code>\skip_gadd:Nn</code>	
<code>\skip_gadd:cn</code>	

Updated: 2011-10-22

<code>\skip_set:Nn</code>	<code>\skip_set:Nn</code> $\langle skip \rangle$ $\{\langle skip\ expression \rangle\}$
<code>\skip_set:cn</code>	Sets $\langle skip \rangle$ to the value of $\langle skip\ expression \rangle$, which must evaluate to a length with units
<code>\skip_gset:Nn</code>	and may include a rubber component (for example 1 cm plus 0.5 cm).
<code>\skip_gset:cn</code>	

Updated: 2011-10-22

<code>\skip_set_eq:NN</code>	<code>\skip_set_eq:NN</code> $\langle skip_1 \rangle$ $\langle skip_2 \rangle$
<code>\skip_set_eq:(cN Nc cc)</code>	Sets the content of $\langle skip_1 \rangle$ equal to that of $\langle skip_2 \rangle$.
<code>\skip_gset_eq:NN</code>	
<code>\skip_gset_eq:(cN Nc cc)</code>	

<code>\skip_sub:Nn</code>	<code>\skip_sub:Nn</code> $\langle skip \rangle$ $\{\langle skip\ expression \rangle\}$
<code>\skip_sub:cn</code>	Subtracts the result of the $\langle skip\ expression \rangle$ from the current content of the $\langle skip \rangle$.
<code>\skip_gsub:Nn</code>	
<code>\skip_gsub:cn</code>	

Updated: 2011-10-22

13 Skip expression conditionals

<code>\skip_if_eq_p:nn</code> ★	<code>\skip_if_eq_p:nn {\langle skipexpr_1 \rangle} {\langle skipexpr_2 \rangle}</code>
<code>\skip_if_eq:nnTF</code> ★	<code>\skip_if_eq:nnTF</code> <code>{\langle skipexpr_1 \rangle} {\langle skipexpr_2 \rangle}</code> <code>{\langle true code \rangle} {\langle false code \rangle}</code>

This function first evaluates each of the $\langle skip\ expressions \rangle$ as described for `\skip_eval:n`. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true.

<code>\skip_if_finite_p:n</code> ★	<code>\skip_if_finite_p:n {\langle skipexpr \rangle}</code>
<code>\skip_if_finite:nTF</code> ★	<code>\skip_if_finite:nTF {\langle skipexpr \rangle} {\langle true code \rangle} {\langle false code \rangle}</code>

New: 2012-03-05

Evaluates the $\langle skip\ expression \rangle$ as described for `\skip_eval:n`, and then tests if all of its components are finite.

14 Using skip expressions and variables

<code>\skip_eval:n</code> ★	<code>\skip_eval:n {\langle skip expression \rangle}</code>
-----------------------------	---

Updated: 2011-10-22

Evaluates the $\langle skip\ expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring `\skip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle glue\ denotation \rangle$ after two expansions. This is expressed in points (pt), and requires suitable termination if used in a T_EX-style assignment as it is *not* an $\langle internal\ glue \rangle$.

<code>\skip_use:N</code> ★	<code>\skip_use:N \langle skip \rangle</code>
<code>\skip_use:c</code> ★	

Recovers the content of a $\langle skip \rangle$ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of `\skip_eval:n`).

T_EXhackers note: `\skip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

15 Viewing skip variables

<code>\skip_show:N</code>	<code>\skip_show:N \langle skip \rangle</code>
<code>\skip_show:c</code>	

Updated: 2015-08-03

Displays the value of the $\langle skip \rangle$ on the terminal.

<code>\skip_show:n</code>	<code>\skip_show:n {\langle skip expression \rangle}</code>
---------------------------	---

New: 2011-11-22

Updated: 2015-08-07

Displays the result of evaluating the $\langle skip\ expression \rangle$ on the terminal.

<code>\skip_log:N</code>	<code>\skip_log:N <skip></code>
<code>\skip_log:c</code>	Writes the value of the <code><skip></code> in the log file.
New: 2014-08-22	
Updated: 2015-08-03	

<code>\skip_log:n</code>	<code>\skip_log:n {\<skip expression>}</code>
	Writes the result of evaluating the <code><skip expression></code> in the log file.
New: 2014-08-22	
Updated: 2015-08-07	

16 Constant skips

<code>\c_max_skip</code>	The maximum value that can be stored as a skip (equal to <code>\c_max_dim</code> in length), with no stretch nor shrink component.
Updated: 2012-11-02	

<code>\c_zero_skip</code>	A zero length as a skip, with no stretch nor shrink component.
Updated: 2012-11-01	

17 Scratch skips

<code>\l_tmpa_skip</code> <code>\l_tmpb_skip</code>	Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	--

<code>\g_tmpa_skip</code> <code>\g_tmpb_skip</code>	Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	---

18 Inserting skips into the output

<code>\skip_horizontal:N</code>	<code>\skip_horizontal:N <skip></code>
<code>\skip_horizontal:c</code>	<code>\skip_horizontal:n {\<skipexpr>}</code>
<code>\skip_horizontal:n</code>	Inserts a horizontal <code><skip></code> into the current list.
Updated: 2011-10-22	
T_EXhackers note: <code>\skip_horizontal:N</code> is the T _E X primitive <code>\hskip</code> renamed.	

<code>\skip_vertical:N</code>	<code>\skip_vertical:N <skip></code>
<code>\skip_vertical:c</code>	<code>\skip_vertical:n {\<skipexpr>}</code>
<code>\skip_vertical:n</code>	Inserts a vertical <code><skip></code> into the current list.
Updated: 2011-10-22	
T_EXhackers note: <code>\skip_vertical:N</code> is the T _E X primitive <code>\vskip</code> renamed.	

19 Creating and initialising muskip variables

`\muskip_new:N`
`\muskip_new:c`

`\muskip_new:N` $\langle muskip \rangle$

Creates a new $\langle muskip \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle muskip \rangle$ is initially equal to 0 mu.

`\muskip_const:Nn`
`\muskip_const:cn`

`\muskip_const:Nn` $\langle muskip \rangle$ $\{ \langle muskip expression \rangle \}$

Creates a new constant $\langle muskip \rangle$ or raises an error if the name is already taken. The value of the $\langle muskip \rangle$ is set globally to the $\langle muskip expression \rangle$.

New: 2012-03-05

`\muskip_zero:N`
`\muskip_zero:c`
`\muskip_gzero:N`
`\muskip_gzero:c`

`\skip_zero:N` $\langle muskip \rangle$

Sets $\langle muskip \rangle$ to 0 mu.

`\muskip_zero_new:N`
`\muskip_zero_new:c`
`\muskip_gzero_new:N`
`\muskip_gzero_new:c`

`\muskip_zero_new:N` $\langle muskip \rangle$

Ensures that the $\langle muskip \rangle$ exists globally by applying `\muskip_new:N` if necessary, then applies `\muskip_(g)zero:N` to leave the $\langle muskip \rangle$ set to zero.

New: 2012-01-07

`\muskip_if_exist_p:N` ★
`\muskip_if_exist_p:c` ★
`\muskip_if_exist:NTF` ★
`\muskip_if_exist:cTF` ★

`\muskip_if_exist_p:N` $\langle muskip \rangle$

`\muskip_if_exist:NTF` $\langle muskip \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$

Tests whether the $\langle muskip \rangle$ is currently defined. This does not check that the $\langle muskip \rangle$ really is a muskip variable.

New: 2012-03-03

20 Setting muskip variables

`\muskip_add:Nn`
`\muskip_add:cn`
`\muskip_gadd:Nn`
`\muskip_gadd:cn`

`\muskip_add:Nn` $\langle muskip \rangle$ $\{ \langle muskip expression \rangle \}$

Adds the result of the $\langle muskip expression \rangle$ to the current content of the $\langle muskip \rangle$.

Updated: 2011-10-22

`\muskip_set:Nn`
`\muskip_set:cn`
`\muskip_gset:Nn`
`\muskip_gset:cn`

`\muskip_set:Nn` $\langle muskip \rangle$ $\{ \langle muskip expression \rangle \}$

Sets $\langle muskip \rangle$ to the value of $\langle muskip expression \rangle$, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).

Updated: 2011-10-22

`\muskip_set_eq:NN`
`\muskip_set_eq:(cN|Nc|cc)`
`\muskip_gset_eq:NN`
`\muskip_gset_eq:(cN|Nc|cc)`

`\muskip_set_eq:NN` $\langle muskip_1 \rangle$ $\langle muskip_2 \rangle$

Sets the content of $\langle muskip_1 \rangle$ equal to that of $\langle muskip_2 \rangle$.

<hr/> <code>\muskip_sub:Nn</code>	<code>\muskip_sub:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_sub:cn</code>	
<code>\muskip_gsub:Nn</code>	Subtracts the result of the <i><muskip expression></i> from the current content of the <i><skip></i> .
<code>\muskip_gsub:cn</code>	
<hr/> Updated: 2011-10-22 <hr/>	

21 Using muskip expressions and variables

<hr/> <code>\muskip_eval:n</code> ★	<code>\muskip_eval:n {<muskip expression>}</code>
<code>\muskip_eval:cn</code>	
<hr/> Updated: 2011-10-22 <hr/>	
	Evaluates the <i><muskip expression></i> , expanding any skips and token list variables within the <i><expression></i> to their content (without requiring <code>\muskip_use:N/\tl_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a <i><mu glue denotation></i> after two expansions. This is expressed in <code>mu</code> , and requires suitable termination if used in a T _E X-style assignment as it is <i>not</i> an <i><internal mu glue></i> .

<hr/> <code>\muskip_use:N</code> ★	<code>\muskip_use:N <muskip></code>
<code>\muskip_use:cn</code> ★	
<hr/>	
	Recovers the content of a <i><skip></i> and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a <i><dimension></i> is required (such as in the argument of <code>\muskip_eval:n</code>).

T_EXhackers note: `\muskip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

22 Viewing muskip variables

<hr/> <code>\muskip_show:N</code>	<code>\muskip_show:N <muskip></code>
<code>\muskip_show:cn</code>	
<hr/> Updated: 2015-08-03 <hr/>	
	Displays the value of the <i><muskip></i> on the terminal.

<hr/> <code>\muskip_show:n</code>	<code>\muskip_show:n {<muskip expression>}</code>
<code>\muskip_show:cn</code>	
<hr/> New: 2011-11-22 <hr/> Updated: 2015-08-07 <hr/>	
	Displays the result of evaluating the <i><muskip expression></i> on the terminal.

<hr/> <code>\muskip_log:N</code>	<code>\muskip_log:N <muskip></code>
<code>\muskip_log:cn</code>	
<hr/> New: 2014-08-22 <hr/> Updated: 2015-08-03 <hr/>	
	Writes the value of the <i><muskip></i> in the log file.

<hr/> <code>\muskip_log:n</code>	<code>\muskip_log:n {<muskip expression>}</code>
<code>\muskip_log:cn</code>	
<hr/> New: 2014-08-22 <hr/> Updated: 2015-08-07 <hr/>	
	Writes the result of evaluating the <i><muskip expression></i> in the log file.

23 Constant muskips

<code>\c_max_muskip</code>	The maximum value that can be stored as a muskip, with no stretch nor shrink component.
----------------------------	---

<code>\c_zero_muskip</code>	A zero length as a muskip, with no stretch nor shrink component.
-----------------------------	--

24 Scratch muskips

<code>\l_tmpa_muskip</code> <code>\l_tmpb_muskip</code>	Scratch muskip for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	--

<code>\g_tmpa_muskip</code> <code>\g_tmpb_muskip</code>	Scratch muskip for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
--	---

25 Primitive conditional

<code>\if_dim:w</code>	<code>\if_dim:w <dimen₁> <relation> <dimen₂></code> <code> <true code></code> <code>\else:</code> <code> <false></code> <code>\fi:</code>
------------------------	---

Compare two dimensions. The *<relation>* is one of <, = or > with category code 12.

T_EXhackers note: This is the T_EX primitive `\ifdim`.

Part XX

The l3keys package

Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

```
\MyModuleSetup{
  key-one = value one,
  key-two = value two
}
```

or

```
\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }
{
  key-one .code:n = code including parameter #1,
  key-two .tl_set:N = \l_mymodule_store_tl
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
{
  key-one = value one,
  key-two = value two
}
```

At a document level, `\keys_set:nn` is used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }
{ \keys_set:nn { mymodule } { #1 } }
\DeclareDocumentCommand \MyModuleMacro { o m }
{
  \group_begin:
    \keys_set:nn { mymodule } { #1 }
    % Main code for \MyModuleMacro
  \group_end:
}
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As discussed in section 2, it is suggested that the character `/` is reserved for sub-division of keys into logical groups. Functions and variables are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
{
  \l_mymodule_tmp_tl .code:n = code
}
```

creates a key called `\l_mymodule_tmp_tl`, and not one called `key`.

1 Creating keys

`\keys_define:nn`

Updated: 2017-11-14

`\keys_define:nn {<module>} {<keyval list>}`

Parses the *<keyval list>* and defines the keys listed there for *<module>*. The *<module>* name should be a text value, but there are no restrictions on the nature of the text. In practice the *<module>* should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The *<keyval list>* should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule }
{
  keyname .code:n = Some-code~using~#1,
  keyname .value_required:n = true
}
```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require one or more arguments. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary *<key>*, which when used may be supplied with a *<value>*. All key *definitions* are local.

Key properties are applied in the reading order and so the ordering is significant. Key properties which define “actions”, such as `.code:n`, `.tl_set:N`, *etc.*, override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so they replace one another. However, properties covering non-exclusive behaviours may be given in any order. Thus for example the following definitions are equivalent.

```
\keys_define:nn { mymodule }
{
  keyname .code:n          = Some-code~using~#1,
  keyname .value_required:n = true
}
\keys_define:nn { mymodule }
```

```

{
  keyname .value_required:n = true,
  keyname .code:n           = Some~code~using~#1
}

```

Note that with the exception of the special `.undefine:` property, all key properties define the key within the current \TeX scope.

```

.bool_set:N
.bool_set:c
.bool_gset:N
.bool_gset:c

```

Updated: 2013-07-08

$\langle key \rangle$.bool_set:N = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to $\langle value \rangle$ (which must be either `true` or `false`). If the variable does not exist, it will be created globally at the point that the key is set up.

```

.bool_set_inverse:N
.bool_set_inverse:c
.bool_gset_inverse:N
.bool_gset_inverse:c

```

New: 2011-08-28

Updated: 2013-07-08

$\langle key \rangle$.bool_set_inverse:N = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to the logical inverse of $\langle value \rangle$ (which must be either `true` or `false`). If the $\langle boolean \rangle$ does not exist, it will be created globally at the point that the key is set up.

```

.choice:

```

$\langle key \rangle$.choice:

Sets $\langle key \rangle$ to act as a choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section 3.

```

.choices:nn
.choices:(Vn|on|xn)

```

New: 2011-08-21

Updated: 2013-07-10

$\langle key \rangle$.choices:nn = $\{\langle choices \rangle\}$ $\{\langle code \rangle\}$

Sets $\langle key \rangle$ to act as a choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 1). Choices are discussed in detail in section 3.

```

.clist_set:N
.clist_set:c
.clist_gset:N
.clist_gset:c

```

New: 2011-09-11

$\langle key \rangle$.clist_set:N = $\langle comma list variable \rangle$

Defines $\langle key \rangle$ to set $\langle comma list variable \rangle$ to $\langle value \rangle$. Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.

```

.code:n

```

Updated: 2013-07-10

$\langle key \rangle$.code:n = $\{\langle code \rangle\}$

Stores the $\langle code \rangle$ for execution when $\langle key \rangle$ is used. The $\langle code \rangle$ can include one parameter (#1), which will be the $\langle value \rangle$ given for the $\langle key \rangle$. The x-type variant expands $\langle code \rangle$ at the point where the $\langle key \rangle$ is created.

`.default:n`
`.default:(V|o|x)`
Updated: 2013-07-09

`<key> .default:n = {<default>}`

Creates a *<default>* value for *<key>*, which is used if no value is given. This will be used if only the key name is given, but not if a blank *<value>* is given:

```
\keys_define:nn { mymodule }
{
  key .code:n      = Hello~#1,
  key .default:n = World
}
\keys_set:nn { mymodule }
{
  key = Fred, % Prints 'Hello Fred'
  key,      % Prints 'Hello World'
  key = ,    % Prints 'Hello '
}
```

The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value does not trigger an error.

`.dim_set:N`
`.dim_set:c`
`.dim_gset:N`
`.dim_gset:c`

`<key> .dim_set:N = <dimension>`

Defines *<key>* to set *<dimension>* to *<value>* (which must a dimension expression). If the variable does not exist, it is created globally at the point that the key is set up.

`.fp_set:N`
`.fp_set:c`
`.fp_gset:N`
`.fp_gset:c`

`<key> .fp_set:N = <floating point>`

Defines *<key>* to set *<floating point>* to *<value>* (which must a floating point expression). If the variable does not exist, it is created globally at the point that the key is set up.

`.groups:n`
New: 2013-07-14

`<key> .groups:n = {<groups>}`

Defines *<key>* as belonging to the *<groups>* declared. Groups provide a “secondary axis” for selectively setting keys, and are described in Section 6.

`.inherit:n`
New: 2016-11-22

`<key> .inherit:n = {<parents>}`

Specifies that the *<key>* path should inherit the keys listed as *<parents>*. For example, after setting

```
\keys_define:n { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:n { } { bar .inherit:n = foo }
```

setting

```
\keys_set:n { bar } { test = a }
```

will be equivalent to

```
\keys_set:n { foo } { test = a }
```


<hr/> <code>.initial:n</code> <hr/>	<code><key> .initial:n = {<value>}</code>
<code>.initial:(V o x)</code>	Initialises the <code><key></code> with the <code><value></code> , equivalent to
<hr/> Updated: 2013-07-09 <hr/>	<code>\keys_set:nn {<module>} { <key> = <value> }</code>
<hr/> <code>.int_set:N</code> <hr/>	<code><key> .int_set:N = <integer></code>
<code>.int_set:c</code>	Defines <code><key></code> to set <code><integer></code> to <code><value></code> (which must be an integer expression). If the variable does not exist, it is created globally at the point that the key is set up.
<code>.int_gset:N</code>	
<code>.int_gset:c</code> <hr/>	
<hr/> <code>.meta:n</code> <hr/>	<code><key> .meta:n = {<keyval list>}</code>
<hr/> Updated: 2013-07-10 <hr/>	Makes <code><key></code> a meta-key, which will set <code><keyval list></code> in one go. The <code><keyval list></code> can refer as <code>#1</code> to the value given at the time the <code><key></code> is used (or, if no value is given, the <code><key></code> 's default value).
<hr/> <code>.meta:nn</code> <hr/>	<code><key> .meta:nn = {<path>} {<keyval list>}</code>
<hr/> New: 2013-07-10 <hr/>	Makes <code><key></code> a meta-key, which will set <code><keyval list></code> in one go using the <code><path></code> in place of the current one. The <code><keyval list></code> can refer as <code>#1</code> to the value given at the time the <code><key></code> is used (or, if no value is given, the <code><key></code> 's default value).
<hr/> <code>.multichoice:</code> <hr/>	<code><key> .multichoice:</code>
<hr/> New: 2011-08-21 <hr/>	Sets <code><key></code> to act as a multiple choice key. Each valid choice for <code><key></code> must then be created, as discussed in section 3.
<hr/> <code>.multichoices:nn</code> <hr/>	<code><key> .multichoices:nn {<choices>} {<code>}</code>
<code>.multichoices:(Vn on xn)</code>	Sets <code><key></code> to act as a multiple choice key, and defines a series <code><choices></code> which are implemented using the <code><code></code> . Inside <code><code></code> , <code>\l_keys_choice_tl</code> will be the name of the choice made, and <code>\l_keys_choice_int</code> will be the position of the choice in the list of <code><choices></code> (indexed from 1). Choices are discussed in detail in section 3.
<hr/> New: 2011-08-21 <hr/>	
<hr/> Updated: 2013-07-10 <hr/>	
<hr/> <code>.skip_set:N</code> <hr/>	<code><key> .skip_set:N = <skip></code>
<code>.skip_set:c</code>	Defines <code><key></code> to set <code><skip></code> to <code><value></code> (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up.
<code>.skip_gset:N</code>	
<code>.skip_gset:c</code> <hr/>	
<hr/> <code>.tl_set:N</code> <hr/>	<code><key> .tl_set:N = <token list variable></code>
<code>.tl_set:c</code>	Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> . If the variable does not exist, it is created globally at the point that the key is set up.
<code>.tl_gset:N</code>	
<code>.tl_gset:c</code> <hr/>	
<hr/> <code>.tl_set_x:N</code> <hr/>	<code><key> .tl_set_x:N = <token list variable></code>
<code>.tl_set_x:c</code>	Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> , which will be subjected to an x-type expansion (<i>i.e.</i> using <code>\tl_set:Nx</code>). If the variable does not exist, it is created globally at the point that the key is set up.
<code>.tl_gset_x:N</code>	
<code>.tl_gset_x:c</code> <hr/>	

<code>.undefine:</code>	<code><key> .undefine:</code>
-------------------------	-------------------------------------

New: 2015-07-14	Removes the definition of the <code><key></code> within the current scope.
-----------------	--

<code>.value_forbidden:n</code>	<code><key> .value_forbidden:n = true false</code>
---------------------------------	--

New: 2015-07-14	Specifies that <code><key></code> cannot receive a <code><value></code> when used. If a <code><value></code> is given then an error will be issued. Setting the property <code>false</code> cancels the restriction.
-----------------	--

<code>.value_required:n</code>	<code><key> .value_required:n = true false</code>
--------------------------------	---

New: 2015-07-14	Specifies that <code><key></code> must receive a <code><value></code> when used. If a <code><value></code> is not given then an error will be issued. Setting the property <code>false</code> cancels the restriction.
-----------------	--

2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { module / subgroup }
{ key .code:n = code }
```

or to the key name:

```
\keys_define:nn { mymodule }
{ subgroup / key .code:n = code }
```

As illustrated, the best choice of token for sub-dividing keys in this way is `/`. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `module/subgroup/key`.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

3 Choice and multiple choice keys

The `l3keys` system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. “Multiple” choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the `.choice:` property:

```
\keys_define:nn { mymodule }
{ key .choice: }
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the `.choices:nn` property.

```

\keys_define:nn { mymodule }
{
  key .choices:nn =
    { choice-a, choice-b, choice-c }
    {
      You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
      which~is~in~position~\int_use:N \l_keys_choice_int \c_space_tl
      in~the~list.
    }
}

```

The index `\l_keys_choice_int` in the list of choices starts at 1.

`\l_keys_choice_int`
`\l_keys_choice_tl`

Inside the code block for a choice generated using `.choices:nn`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using `.code:n`, the value passed to the key (i.e. the choice name) is also available as `#1`.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```

\keys_define:nn { mymodule }
{
  key .choice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}

```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`. These variables do not have defined behaviour when used outside of code created using `.choices:nn` (i.e. anything might happen).

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special `unknown` choice. The general behavior of the `unknown` key is described in Section 5. A typical example in the case of a choice would be to issue a custom error message:

```

\keys_define:nn { mymodule }
{
  key .choice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
  key / unknown .code:n =
    \msg_error:nnxxx { mymodule } { unknown-choice }
    { key } % Name of choice key
    { choice-a , choice-b , choice-c } % Valid choices
    { \exp_not:n {#1} } % Invalid choice given
}

```

```

%
%
}

```

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are define as sub-keys. Thus both

```

\keys_define:nn { mymodule }
{
  key .multichoices:nn =
    { choice-a, choice-b, choice-c }
    {
      You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
      which~is~in~position~
      \int_use:N \l_keys_choice_int \c_space_tl
      in~the~list.
    }
}

```

and

```

\keys_define:nn { mymodule }
{
  key .multichoice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}

```

are valid.

When a multiple choice key is set

```

\keys_set:nn { mymodule }
{
  key = { a , b , c } % 'key' defined as a multiple choice
}

```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```

\keys_set:nn { mymodule }
{
  key = a ,
  key = b ,
  key = c ,
}

```

Thus each separate choice will have passed to it the `\l_keys_choice_tl` and `\l_keys_choice_int` in exactly the same way as described for `.choices:nn`.

4 Setting keys

```
\keys_set:nn
\keys_set:(nV|nv|no)
```

Updated: 2017-11-14

```
\keys_set:nn {<module>} {<keyval list>}
```

Parses the *<keyval list>*, and sets those keys which are defined for *<module>*. The behaviour on finding an unknown key can be set by defining a special **unknown** key: this is illustrated later.

```
\l_keys_key_tl
\l_keys_path_tl
\l_keys_value_tl
```

Updated: 2015-07-14

For each key processed, information of the full *path* of the key, the *name* of the key and the *value* of the key is available within three token list variables. These may be used within the code of the key.

The *value* is everything after the =, which may be empty if no value was given. This is stored in `\l_keys_value_tl`, and is not processed in any way by `\keys_set:nn`.

The *path* of the key is a “full” description of the key, and is unique for each key. It consists of the module and full key name, thus for example

```
\keys_set:nn { mymodule } { key-a = some-value }
```

has path `mymodule/key-a` while

```
\keys_set:nn { mymodule } { subset / key-a = some-value }
```

has path `mymodule/subset/key-a`. This information is stored in `\l_keys_path_tl`, and will have been processed by `\tl_to_str:n`.

The *name* of the key is the part of the path after the last /, and thus is not unique. In the preceding examples, both keys have name `key-a` despite having different paths. This information is stored in `\l_keys_key_tl`, and will have been processed by `\tl_to_str:n`.

5 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` looks for a special **unknown** key for the same module, and if this is not defined raises an error indicating that the key name was unknown. This mechanism can be used for example to issue custom error texts.

```
\keys_define:nn { mymodule }
{
  unknown .code:n =
    You~tried~to~set~key~'\l_keys_key_tl'~to~'#1'.
}
```

```

\keys_set_known:nnN      \keys_set_known:nnN {<module>} {<keyval list>} {<tl>}
\keys_set_known:(nVN|nvN|noN)
\keys_set_known:nn
\keys_set_known:(nV|nv|no)

```

New: 2011-08-23
Updated: 2017-05-27

In some cases, the desired behavior is to simply ignore unknown keys, collecting up information on these for later processing. The `\keys_set_known:nnN` function parses the `<keyval list>`, and sets those keys which are defined for `<module>`. Any keys which are unknown are not processed further by the parser. The key-value pairs for each *unknown* key name are stored in the `<tl>` in a comma-separated form (*i.e.* an edited version of the `<keyval list>`). The `\keys_set_known:nn` version skips this stage.

Use of `\keys_set_known:nnN` can be nested, with the correct residual `<keyval list>` returned at each stage.

6 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

```

\keys define:nn { mymodule }
{
  key-one   .code:n   = { \my_func:n {#1} } ,
  key-two   .tl_set:N = \l_my_a_tl          ,
  key-three .tl_set:N = \l_my_b_tl          ,
  key-four  .fp_set:N = \l_my_a_fp          ,
}

```

the use of `\keys_set:nn` attempts to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```

\keys define:nn { mymodule }
{
  key-one   .code:n   = { \my_func:n {#1} } ,
  key-one   .groups:n = { first }           ,
  key-two   .tl_set:N = \l_my_a_tl          ,
  key-two   .groups:n = { first }           ,
  key-three .tl_set:N = \l_my_b_tl          ,
  key-three .groups:n = { second }          ,
  key-four  .fp_set:N = \l_my_a_fp          ,
}

```

assigns `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made “active”, or by marking one or more groups to be ignored in key setting.

<code>\keys_set_filter:nnnN</code>	<code>\keys_set_filter:nnnN {<module>} {<groups>} {<keyval list>} <tl></code>
<code>\keys_set_filter:(nnVN nnvN nnoN)</code>	
<code>\keys_set_filter:nnn</code>	
<code>\keys_set_filter:(nnV nnv nno)</code>	

New: 2013-07-14

Updated: 2017-05-27

Activates key filtering in an “opt-out” sense: keys assigned to any of the $\langle groups \rangle$ specified are ignored. The $\langle groups \rangle$ are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set. The key-value pairs for each key which is filtered out are stored in the $\langle tl \rangle$ in a comma-separated form (*i.e.* an edited version of the $\langle keyval list \rangle$). The `\keys_set_filter:nnn` version skips this stage.

Use of `\keys_set_filter:nnnN` can be nested, with the correct residual $\langle keyval list \rangle$ returned at each stage.

<code>\keys_set_groups:nnn</code>	<code>\keys_set_groups:nnn {<module>} {<groups>} {<keyval list>}</code>
<code>\keys_set_groups:(nnV nnv nno)</code>	

New: 2013-07-14

Updated: 2017-05-27

Activates key filtering in an “opt-in” sense: only keys assigned to one or more of the $\langle groups \rangle$ specified are set. The $\langle groups \rangle$ are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set.

7 Utility functions for keys

<code>\keys_if_exist_p:nn</code> ★	<code>\keys_if_exist_p:nn {<module>} {<key>}</code>
<code>\keys_if_exist:nnTF</code> ★	<code>\keys_if_exist:nnTF {<module>} {<key>} {<true code>} {<false code>}</code>

Updated: 2017-11-14 Tests if the $\langle key \rangle$ exists for $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle$.

<code>\keys_if_choice_exist_p:nnn</code> ★	<code>\keys_if_choice_exist_p:nnn {<module>} {<key>} {<choice>}</code>
<code>\keys_if_choice_exist:nnnTF</code> ★	<code>\keys_if_choice_exist:nnnTF {<module>} {<key>} {<choice>} {<true code>} {<false code>}</code>

New: 2011-08-21

Updated: 2017-11-14

Tests if the $\langle choice \rangle$ is defined for the $\langle key \rangle$ within the $\langle module \rangle$, *i.e.* if any code has been defined for $\langle key \rangle / \langle choice \rangle$. The test is **false** if the $\langle key \rangle$ itself is not defined.

<code>\keys_show:nn</code>	<code>\keys_show:nn {<module>} {<key>}</code>
----------------------------	---

Updated: 2015-08-09

Displays in the terminal the information associated to the $\langle key \rangle$ for a $\langle module \rangle$, including the function which is used to actually implement it.

<code>\keys_log:nn</code>	<code>\keys_log:nn {<module>} {<key>}</code>
---------------------------	--

New: 2014-08-22

Updated: 2015-08-09

Writes in the log file the information associated to the $\langle key \rangle$ for a $\langle module \rangle$. See also `\keys_show:nn` which displays the result in the terminal.

8 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,  
KeyTwo = ValueTwo ,  
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in special circumstances you may wish to use a lower-level approach. The low-level parsing system converts a *key–value list* into *keys* and associated *values*. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double # tokens or expand any input. Active tokens = and , appearing at the outer level of braces are converted to category “other” (12) so that the parser does not “miss” any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

\keyval_parse:NNn

Updated: 2011-09-08

\keyval_parse:NNn $\langle function_1 \rangle$ $\langle function_2 \rangle$ { $\langle key-value list \rangle$ }

Parses the $\langle key-value list \rangle$ into a series of $\langle keys \rangle$ and associated $\langle values \rangle$, or keys alone (if no $\langle value \rangle$ was given). $\langle function_1 \rangle$ should take one argument, while $\langle function_2 \rangle$ should absorb two arguments. After **\keyval_parse:NNn** has parsed the $\langle key-value list \rangle$, $\langle function_1 \rangle$ is used to process keys given with no value and $\langle function_2 \rangle$ is used to process keys given with a value. The order of the $\langle keys \rangle$ in the $\langle key-value list \rangle$ is preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the $\langle key \rangle$ and $\langle value \rangle$, then one *outer* set of braces is removed from the $\langle key \rangle$ and $\langle value \rangle$ as part of the processing.

Part XXI

The l3fp package: Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x / y , square root \sqrt{x} , and parentheses.
- Comparison operators: $x < y$, $x \leq y$, $x > y$, $x \neq y$ etc.
- Boolean logic: sign $\text{sign } x$, negation $!x$, conjunction $x \&\& y$, disjunction $x || y$, ternary operator $x ? y : z$.
- Exponentials: $\exp x$, $\ln x$, x^y .
- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\text{sind } x$, $\text{cosd } x$, $\text{tand } x$, $\text{cotd } x$, $\text{secd } x$, $\text{cscd } x$ expecting their arguments in degrees.
- Inverse trigonometric functions: $\text{asin } x$, $\text{acos } x$, $\text{atan } x$, $\text{acot } x$, $\text{asec } x$, $\text{acsc } x$ giving a result in radians, and $\text{asind } x$, $\text{acosd } x$, $\text{atand } x$, $\text{acotd } x$, $\text{asecd } x$, $\text{acscd } x$ giving a result in degrees.

(*not yet*) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\text{sech } x$, $\text{csch } x$, and $\text{asinh } x$, $\text{acosh } x$, $\text{atanh } x$, $\text{acoth } x$, $\text{asech } x$, $\text{acsch } x$.

- Extrema: $\max(x, y, \dots)$, $\min(x, y, \dots)$, $\text{abs}(x)$.
- Rounding functions ($n = 0$ by default, $t = \text{NaN}$ by default): $\text{trunc}(x, n)$ rounds towards zero, $\text{floor}(x, n)$ rounds towards $-\infty$, $\text{ceil}(x, n)$ rounds towards $+\infty$, $\text{round}(x, n, t)$ rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$. And (*not yet*) modulo, and “quantize”.
- Random numbers: $\text{rand}()$, $\text{randint}(m, n)$ in all engines except X_YT_EX.
- Constants: **pi**, **deg** (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, **pc** is 12.
- Automatic conversion (no need for $\backslash\langle\text{type}\rangle_use:N$) of integer, dimension, and skip variables to floating point numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
- Tuples: (x_1, \dots, x_n) that can be stored in variables, added together, multiplied or divided by a floating point number, and nested.

Floating point numbers can be given either explicitly (in a form such as $1.234\text{e-}34$, or $-.0001$), or as a stored floating point variable, which is automatically replaced by its current value. A “floating point” is a floating point number or a tuple thereof. See section 9.1 for a description of what a floating point is, section 9.2 for details about how an expression is parsed, and section 9.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 7.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}$
= \ExplSyntaxOn \fp_to_decimal:n {\sin(3.5)/2 + 2e-3} $.
```

The operation `round` can be used to limit the result’s precision. Adding `+0` avoids the possibly undesirable output `-0`, replacing it by `+0`. However, the `l3fp` module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\documentclass{article}
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calculus } { m }
{ \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calculus { 2 pi * sin ( 2.3 ^ 5 ) }
\end{document}
```

See the documentation of `siunitx` for various options of `\num`.

1 Creating and initialising floating point variables

<code>\fp_new:N</code> <code>\fp_new:c</code> Updated: 2012-05-08	<code>\fp_new:N <fp var></code> Creates a new <code><fp var></code> or raises an error if the name is already taken. The declaration is global. The <code><fp var></code> is initially <code>+0</code> .
<code>\fp_const:Nn</code> <code>\fp_const:cn</code> Updated: 2012-05-08	<code>\fp_const:Nn <fp var> {<floating point expression>}</code> Creates a new constant <code><fp var></code> or raises an error if the name is already taken. The <code><fp var></code> is set globally equal to the result of evaluating the <code><floating point expression></code> .
<code>\fp_zero:N</code> <code>\fp_zero:c</code> <code>\fp_gzero:N</code> <code>\fp_gzero:c</code> Updated: 2012-05-08	<code>\fp_zero:N <fp var></code> Sets the <code><fp var></code> to <code>+0</code> .
<code>\fp_zero_new:N</code> <code>\fp_zero_new:c</code> <code>\fp_gzero_new:N</code> <code>\fp_gzero_new:c</code> Updated: 2012-05-08	<code>\fp_zero_new:N <fp var></code> Ensures that the <code><fp var></code> exists globally by applying <code>\fp_new:N</code> if necessary, then applies <code>\fp_(g)zero:N</code> to leave the <code><fp var></code> set to <code>+0</code> .

2 Setting floating point variables

<code>\fp_set:Nn</code>	<code>\fp_set:Nn <fp var> {(floating point expression)}</code>
<code>\fp_set:cn</code>	
<code>\fp_gset:Nn</code>	Sets $\langle fp\ var \rangle$ equal to the result of computing the $\langle floating\ point\ expression \rangle$.
<code>\fp_gset:cn</code>	

Updated: 2012-05-08

<code>\fp_set_eq:NN</code>	<code>\fp_set_eq:NN <fp var₁> <fp var₂></code>
<code>\fp_set_eq:(cN Nc cc)</code>	
<code>\fp_gset_eq:NN</code>	Sets the floating point variable $\langle fp\ var_1 \rangle$ equal to the current value of $\langle fp\ var_2 \rangle$.
<code>\fp_gset_eq:(cN Nc cc)</code>	

Updated: 2012-05-08

<code>\fp_add:Nn</code>	<code>\fp_add:Nn <fp var> {(floating point expression)}</code>
<code>\fp_add:cn</code>	
<code>\fp_gadd:Nn</code>	Adds the result of computing the $\langle floating\ point\ expression \rangle$ to the $\langle fp\ var \rangle$. This also
<code>\fp_gadd:cn</code>	applies if $\langle fp\ var \rangle$ and $\langle floating\ point\ expression \rangle$ evaluate to tuples of the same size.

Updated: 2012-05-08

<code>\fp_sub:Nn</code>	<code>\fp_sub:Nn <fp var> {(floating point expression)}</code>
<code>\fp_sub:cn</code>	
<code>\fp_gsub:Nn</code>	Subtracts the result of computing the $\langle floating\ point\ expression \rangle$ from the $\langle fp\ var \rangle$. This
<code>\fp_gsub:cn</code>	also applies if $\langle fp\ var \rangle$ and $\langle floating\ point\ expression \rangle$ evaluate to tuples of the same size.

Updated: 2012-05-08

3 Using floating points

<code>\fp_eval:n</code> ★	<code>\fp_eval:n {(floating point expression)}</code>
New: 2012-05-08	
Updated: 2012-07-08	Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. For a tuple, each item is converted using <code>\fp_eval:n</code> and they are combined as $(\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items. This function is identical to <code>\fp_to_decimal:n</code> .

<code>\fp_to_decimal:N</code> ★	<code>\fp_to_decimal:N <fp var></code>
<code>\fp_to_decimal:c</code> ★	<code>\fp_to_decimal:n {(floating point expression)}</code>
<code>\fp_to_decimal:n</code> ★	Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. For a tuple, each item is converted using <code>\fp_to_decimal:n</code> and they are combined as $(\langle fp_1 \rangle, \sqcup \langle fp_2 \rangle, \sqcup \dots \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.
New: 2012-05-08	
Updated: 2012-07-08	

<code>\fp_to_dim:N</code>	★	<code>\fp_to_dim:N <fp var></code>
<code>\fp_to_dim:c</code>	★	<code>\fp_to_dim:n {<floating point expression>}</code>
<code>\fp_to_dim:n</code>	★	Evaluates the <i><floating point expression></i> and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to <code>\fp_to_decimal:n</code> , with an additional trailing pt (both letter tokens). In particular, the result may be outside the range $[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]$ of valid \TeX dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values $\pm\infty$ and NaN , trigger an “invalid operation” exception.

Updated: 2016-03-22

<code>\fp_to_int:N</code>	★	<code>\fp_to_int:N <fp var></code>
<code>\fp_to_int:c</code>	★	<code>\fp_to_int:n {<floating point expression>}</code>
<code>\fp_to_int:n</code>	★	Evaluates the <i><floating point expression></i> , and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31} + 1, 2^{31} - 1]$ of valid \TeX integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values $\pm\infty$ and NaN , trigger an “invalid operation” exception.

Updated: 2012-07-08

<code>\fp_to_scientific:N</code>	★	<code>\fp_to_scientific:N <fp var></code>
<code>\fp_to_scientific:c</code>	★	<code>\fp_to_scientific:n {<floating point expression>}</code>
<code>\fp_to_scientific:n</code>	★	Evaluates the <i><floating point expression></i> and expresses the result in scientific notation:

New: 2012-05-08
Updated: 2016-03-22

<optional -><digit>.<15 digits>e<optional sign><exponent>

The leading *<digit>* is non-zero except in the case of ± 0 . The values $\pm\infty$ and **NaN** trigger an “invalid operation” exception. Normal category codes apply: thus the **e** is category code 11 (a letter). For a tuple, each item is converted using `\fp_to_scientific:n` and they are combined as $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.

<code>\fp_to_tl:N</code>	★	<code>\fp_to_tl:N <fp var></code>
<code>\fp_to_tl:c</code>	★	<code>\fp_to_tl:n {<floating point expression>}</code>
<code>\fp_to_tl:n</code>	★	Evaluates the <i><floating point expression></i> and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from <code>\fp_to_scientific:n</code>). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see <code>\fp_to_decimal:n</code>). Negative numbers start with - . The special values ± 0 , $\pm\infty$ and NaN are rendered as 0 , -0 , inf , -inf , and nan respectively. Normal category codes apply and thus inf or nan , if produced, are made up of letters. For a tuple, each item is converted using <code>\fp_to_tl:n</code> and they are combined as $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.

Updated: 2016-03-22

<code>\fp_use:N</code>	★	<code>\fp_use:N <fp var></code>
<code>\fp_use:c</code>	★	Inserts the value of the <i><fp var></i> into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and NaN trigger an “invalid operation” exception. For a tuple, each item is converted using <code>\fp_to_decimal:n</code> and they are combined as $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items. This function is identical to <code>\fp_to_decimal:N</code> .

Updated: 2012-07-08

4 Floating point conditionals

<code>\fp_if_exist_p:N</code> ★	<code>\fp_if_exist_p:N <fp var></code>
<code>\fp_if_exist_p:c</code> ★	<code>\fp_if_exist:NTF <fp var> {<true code>} {<false code>}</code>
<code>\fp_if_exist:NTF</code> ★	Tests whether the <i><fp var></i> is currently defined. This does not check that the <i><fp var></i> really is a floating point variable.
<code>\fp_if_exist:cTF</code> ★	

Updated: 2012-05-08

<code>\fp_compare_p:nNn</code> ★	<code>\fp_compare_p:nNn {<fpexpr₁>} <relation> {<fpexpr₂>}</code>
<code>\fp_compare:nNnTF</code> ★	<code>\fp_compare:nNnTF {<fpexpr₁>} <relation> {<fpexpr₂>} {<true code>} {<false code>}</code>

Updated: 2012-05-08

Compares the *<fpexpr₁>* and the *<fpexpr₂>*, and returns **true** if the *<relation>* is obeyed. Two floating points *x* and *y* may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ (“not ordered”). The last case occurs exactly if one or both operands is **NaN** or is a tuple, unless they are equal tuples. Note that a **NaN** is distinct from any value, even another **NaN**, hence $x = x$ is not true for a **NaN**. To test if a value is **NaN**, compare it to an arbitrary number with the “not ordered” relation.

```

\fp_compare:nNnTF { <value> } ? { 0 }
{ } % <value> is nan
{ } % <value> is not nan

```

Tuples are equal if they have the same number of items and items compare equal (in particular there must be no **NaN**). At present any other comparison with tuples yields ? (not ordered). This is experimental.

<code>\fp_compare_p:n</code> ☆	<code>\fp_compare_p:n</code>
<code>\fp_compare:nTF</code> ☆	{
Updated: 2013-12-14	$\langle fpexpr_1 \rangle$ $\langle relation_1 \rangle$
	...
	$\langle fpexpr_N \rangle$ $\langle relation_N \rangle$
	$\langle fpexpr_{N+1} \rangle$
	}
	<code>\fp_compare:nTF</code>
	{
	$\langle fpexpr_1 \rangle$ $\langle relation_1 \rangle$
	...
	$\langle fpexpr_N \rangle$ $\langle relation_N \rangle$
	$\langle fpexpr_{N+1} \rangle$
	}
	{ $\langle true\ code \rangle$ } { $\langle false\ code \rangle$ }

Evaluates the $\langle floating\ point\ expressions \rangle$ as described for `\fp_eval:n` and compares consecutive result using the corresponding $\langle relation \rangle$, namely it compares $\langle intexpr_1 \rangle$ and $\langle intexpr_2 \rangle$ using the $\langle relation_1 \rangle$, then $\langle intexpr_2 \rangle$ and $\langle intexpr_3 \rangle$ using the $\langle relation_2 \rangle$, until finally comparing $\langle intexpr_N \rangle$ and $\langle intexpr_{N+1} \rangle$ using the $\langle relation_N \rangle$. The test yields **true** if all comparisons are **true**. Each $\langle floating\ point\ expression \rangle$ is evaluated only once. Contrarily to `\int_compare:nTF`, all $\langle floating\ point\ expressions \rangle$ are computed, even if one comparison is **false**. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ (“not ordered”). The last case occurs exactly if one or both operands is NaN or is a tuple, unless they are equal tuples. Each $\langle relation \rangle$ can be any (non-empty) combination of $<$, $=$, $>$, and $?$, plus an optional leading $!$ (which negates the $\langle relation \rangle$), with the restriction that the $\langle relation \rangle$ may not start with $?$, as this symbol has a different meaning (in combination with $:$) within floating point expressions. The comparison $x \langle relation \rangle y$ is then **true** if the $\langle relation \rangle$ does not start with $!$ and the actual relation ($<$, $=$, $>$, or $?$) between x and y appears within the $\langle relation \rangle$, or on the contrary if the $\langle relation \rangle$ starts with $!$ and the relation between x and y does not appear within the $\langle relation \rangle$. Common choices of $\langle relation \rangle$ include \geq (greater or equal), \neq (not equal), $!?$ or \leq (comparable).

5 Floating point expression loops

<code>\fp_do_until:nNnn</code> ☆	<code>\fp_do_until:nNnn {$\langle fpexpr_1 \rangle$} $\langle relation \rangle$ {$\langle fpexpr_2 \rangle$} {$\langle code \rangle$}</code>
New: 2012-08-16	Places the $\langle code \rangle$ in the input stream for T _E X to process, and then evaluates the relationship between the two $\langle floating\ point\ expressions \rangle$ as described for <code>\fp_compare:nNnTF</code> . If the test is false then the $\langle code \rangle$ is inserted into the input stream again and a loop occurs until the $\langle relation \rangle$ is true .
<code>\fp_do_while:nNnn</code> ☆	<code>\fp_do_while:nNnn {$\langle fpexpr_1 \rangle$} $\langle relation \rangle$ {$\langle fpexpr_2 \rangle$} {$\langle code \rangle$}</code>
New: 2012-08-16	Places the $\langle code \rangle$ in the input stream for T _E X to process, and then evaluates the relationship between the two $\langle floating\ point\ expressions \rangle$ as described for <code>\fp_compare:nNnTF</code> . If the test is true then the $\langle code \rangle$ is inserted into the input stream again and a loop occurs until the $\langle relation \rangle$ is false .

<hr/> <code>\fp_until_do:nNnn</code> ☆ <hr/>	<code>\fp_until_do:nNnn {<fpexpr1>} <relation> {<fpexpr2>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\fp_while_do:nNnn</code> ☆ <hr/>	<code>\fp_while_do:nNnn {<fpexpr1>} <relation> {<fpexpr2>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .
<hr/> <code>\fp_do_until:nn</code> ☆ <hr/>	<code>\fp_do_until:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is false then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is true .
<hr/> <code>\fp_do_while:nn</code> ☆ <hr/>	<code>\fp_do_while:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> . If the test is true then the <i><code></i> is inserted into the input stream again and a loop occurs until the <i><relation></i> is false .
<hr/> <code>\fp_until_do:nn</code> ☆ <hr/>	<code>\fp_until_do:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is true .
<hr/> <code>\fp_while_do:nn</code> ☆ <hr/>	<code>\fp_while_do:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16 Updated: 2013-12-14	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test is repeated, and a loop occurs until the test is false .

`\fp_step_function:nnnN` ☆
`\fp_step_function:nnnc` ☆

New: 2016-11-21
Updated: 2016-12-06

`\fp_step_function:nnnN` {*⟨initial value⟩*} {*⟨step⟩*} {*⟨final value⟩*} *⟨function⟩*

This function first evaluates the *⟨initial value⟩*, *⟨step⟩* and *⟨final value⟩*, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The *⟨function⟩* is then placed in front of each *⟨value⟩* from the *⟨initial value⟩* to the *⟨final value⟩* in turn (using *⟨step⟩* between each *⟨value⟩*). The *⟨step⟩* must be non-zero. If the *⟨step⟩* is positive, the loop stops when the *⟨value⟩* becomes larger than the *⟨final value⟩*. If the *⟨step⟩* is negative, the loop stops when the *⟨value⟩* becomes smaller than the *⟨final value⟩*. The *⟨function⟩* should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
```

would print

```
[I saw 1.0]   [I saw 1.1]   [I saw 1.2]   [I saw 1.3]   [I saw 1.4]   [I saw 1.5]
```

TpXhackers note: Due to rounding, it may happen that adding the *⟨step⟩* to the *⟨value⟩* does not change the *⟨value⟩*; such cases give an error, as they would otherwise lead to an infinite loop.

`\fp_step_inline:nnnn`

New: 2016-11-21
Updated: 2016-12-06

`\fp_step_inline:nnnn` {*⟨initial value⟩*} {*⟨step⟩*} {*⟨final value⟩*} {*⟨code⟩*}

This function first evaluates the *⟨initial value⟩*, *⟨step⟩* and *⟨final value⟩*, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each *⟨value⟩* from the *⟨initial value⟩* to the *⟨final value⟩* in turn (using *⟨step⟩* between each *⟨value⟩*), the *⟨code⟩* is inserted into the input stream with `#1` replaced by the current *⟨value⟩*. Thus the *⟨code⟩* should define a function of one argument (`#1`).

`\fp_step_variable:nnnNn`

New: 2017-04-12

`\fp_step_variable:nnnNn`
{*⟨initial value⟩*} {*⟨step⟩*} {*⟨final value⟩*} *⟨tl var⟩* {*⟨code⟩*}

This function first evaluates the *⟨initial value⟩*, *⟨step⟩* and *⟨final value⟩*, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each *⟨value⟩* from the *⟨initial value⟩* to the *⟨final value⟩* in turn (using *⟨step⟩* between each *⟨value⟩*), the *⟨code⟩* is inserted into the input stream, with the *⟨tl var⟩* defined as the current *⟨value⟩*. Thus the *⟨code⟩* should make use of the *⟨tl var⟩*.

6 Some useful constants, and scratch variables

`\c_zero_fp`
`\c_minus_zero_fp`

New: 2012-05-08

Zero, with either sign.

`\c_one_fp`

New: 2012-05-08

One as an fp: useful for comparisons in some places.

<hr/> <code>\c_inf_fp</code> <code>\c_minus_inf_fp</code> <hr/>	Infinity, with either sign. These can be input directly in a floating point expression as <code>inf</code> and <code>-inf</code> .
<hr/> New: 2012-05-08 <hr/>	
<hr/> <code>\c_e_fp</code> <hr/>	The value of the base of the natural logarithm, $e = \exp(1)$.
<hr/> Updated: 2012-05-08 <hr/>	
<hr/> <code>\c_pi_fp</code> <hr/>	The value of π . This can be input directly in a floating point expression as <code>pi</code> .
<hr/> Updated: 2013-11-17 <hr/>	
<hr/> <code>\c_one_degree_fp</code> <hr/>	The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as <code>deg</code> .
<hr/> New: 2012-05-08 Updated: 2013-11-17 <hr/>	
<hr/> <code>\l_tmpa_fp</code> <code>\l_tmpb_fp</code> <hr/>	Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_fp</code> <code>\g_tmpb_fp</code> <hr/>	Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

7 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

“Exceptions” may occur when performing some floating point operations, such as `0 / 0`, or `10 ** 1e9999`. The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0 .
- *Invalid operation* occurs for operations with no defined outcome, for instance `0/0` or `sin(∞)`, and results in a NaN. It also occurs for conversion functions whose target type does not have the appropriate infinite or NaN value (*e.g.*, `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating functions at poles, *e.g.*, `ln(0)` or `cot(0)`. This results in $\pm\infty$.

(*not yet*) *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in L^AT_EX3.

To each exception we associate a “flag”: `fp_overflow`, `fp_underflow`, `fp_invalid_operation` and `fp_division_by_zero`. The state of these flags can be tested and modified with commands from `l3flag`

By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

<code>\fp_trap:nn</code>	<code>\fp_trap:nn {<exception>} {<trap type>}</code>
New: 2012-07-19 Updated: 2017-02-13	All occurrences of the <code><exception></code> (<code>overflow</code> , <code>underflow</code> , <code>invalid_operation</code> or <code>division_by_zero</code>) within the current group are treated as <code><trap type></code> , which can be

- **none**: the `<exception>` will be entirely ignored, and leave no trace;
- **flag**: the `<exception>` will turn the corresponding flag on when it occurs;
- **error**: additionally, the `<exception>` will halt the T_EX run and display some information about the current operation in the terminal.

This function is experimental, and may be altered or removed.

```
flag_fp_overflow
flag_fp_underflow
flag_fp_invalid_operation
flag_fp_division_by_zero
```

Flags denoting the occurrence of various floating-point exceptions.

8 Viewing floating points

<code>\fp_show:N</code>	<code>\fp_show:N <fp var></code>
<code>\fp_show:c</code>	<code>\fp_show:n {<floating point expression>}</code>
<code>\fp_show:n</code>	Evaluates the <code><floating point expression></code> and displays the result in the terminal.
New: 2012-05-08 Updated: 2015-08-07	

<code>\fp_log:N</code>	<code>\fp_log:N <fp var></code>
<code>\fp_log:c</code>	<code>\fp_log:n {<floating point expression>}</code>
<code>\fp_log:n</code>	Evaluates the <code><floating point expression></code> and writes the result in the log file.
New: 2014-08-22 Updated: 2015-08-07	

9 Floating point expressions

9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm m \cdot 10^n$, a floating point number, with integer $1 \leq m \leq 10^{16}$, and $-10000 \leq n \leq 10000$;

- ± 0 , zero, with a given sign;
- $\pm \infty$, infinity, with a given sign;
- NaN, is “not a number”, and can be either quiet or signalling (*not yet*: this distinction is currently unsupported);

Normal floating point numbers are stored in base 10, with up to 16 significant figures.

On input, a normal floating point number consists of:

- $\langle sign \rangle$: a possibly empty string of + and - characters;
- $\langle significand \rangle$: a non-empty string of digits together with zero or one dot;
- $\langle exponent \rangle$ optionally: the character **e**, followed by a possibly empty string of + and - tokens, and a non-empty string of digits.

The sign of the resulting number is + if $\langle sign \rangle$ contains an even number of -, and - otherwise, hence, an empty $\langle sign \rangle$ denotes a non-negative input. The stored significand is obtained from $\langle significand \rangle$ by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input $\langle significand \rangle$ has at most 16 digits. The stored $\langle exponent \rangle$ is obtained by combining the input $\langle exponent \rangle$ (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting $\langle exponent \rangle$ is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm \infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if $\langle significand \rangle$ contains no non-zero digit (*i.e.*, consists only in characters 0, and an optional period), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to +0, but that is not guaranteed to remain true.

The $\langle significand \rangle$ must be non-empty, so **e1** and **e-1** are not valid floating point numbers. Note that the latter could be mistaken with the difference of “e” and 1. To avoid confusions, the base of natural logarithms cannot be input as **e** and should be input as **exp(1)** or **\c_e_fp**.

Special numbers are input as follows:

- **inf** represents $+\infty$, and can be preceded by any $\langle sign \rangle$, yielding $\pm \infty$ as appropriate.
- **nan** represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a NaN.
- Note that commands such as **\infty**, **\pi**, or **\sin** *do not* work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.

9.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Binary `*`, `/`, and implicit multiplication by juxtaposition (`2pi`, `3(4+5)`, *etc*).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc*.
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).
- Comma (to build tuples).

The precedence of operations can be overridden using parentheses. In particular, those precedences imply that

$$\begin{aligned}\text{sin2pi} &= \sin(2)\pi! = 0, \\ 2^{\text{2max}(3,5)} &= 2^2 \max(3,5) = 20.\end{aligned}$$

Functions are called on the value of their argument, contrarily to `TeX` macros.

9.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is `false` if it is ± 0 , and `true` otherwise, including when it is `NaN` or a tuple such as `(0,0)`. Tuples are only supported to some extent by operations that work with truth values (`?:`, `||`, `&&`, `!`), by comparisons (`!<=>?`), and by `+`, `-`, `*`, `/`. Unless otherwise specified, providing a tuple as an argument of any other operation yields the “invalid operation” exception and a `NaN` result.

```
?: \fp_eval:n { <operand1> ? <operand2> : <operand3> }
```

The ternary operator `?:` results in $\langle operand_2 \rangle$ if $\langle operand_1 \rangle$ is true (not ± 0), and $\langle operand_3 \rangle$ if $\langle operand_1 \rangle$ is false (± 0). All three $\langle operands \rangle$ are evaluated in all cases; they may be tuples. The operator is right associative, hence

```
\fp_eval:n
{
  1 + 3 > 4 ? 1 :
  2 + 4 > 5 ? 2 :
  3 + 5 > 6 ? 3 : 4
}
```

first tests whether $1 + 3 > 4$; since this isn't true, the branch following `:` is taken, and $2 + 4 > 5$ is compared; since this is true, the branch before `:` is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

```
|| \fp_eval:n { <operand1> || <operand2> }
```

If $\langle operand_1 \rangle$ is true (not ± 0), use that value, otherwise the value of $\langle operand_2 \rangle$. Both $\langle operands \rangle$ are evaluated in all cases; they may be tuples. In $\langle operand_1 \rangle || \langle operand_2 \rangle || \dots || \langle operand_n \rangle$, the first true (nonzero) $\langle operand \rangle$ is used and if all are zero the last one (± 0) is used.

```
&& \fp_eval:n { <operand1> && <operand2> }
```

If $\langle operand_1 \rangle$ is false (equal to ± 0), use that value, otherwise the value of $\langle operand_2 \rangle$. Both $\langle operands \rangle$ are evaluated in all cases; they may be tuples. In $\langle operand_1 \rangle \&\& \langle operand_2 \rangle \&\& \dots \&\& \langle operand_n \rangle$, the first false (± 0) $\langle operand \rangle$ is used and if none is zero the last one is used.

```
< \fp_eval:n
= {
>   <operand1> <relation1>
?   ...
    <operand_N> <relation_N>
Updated: 2013-12-14   <operand_{N+1}>
                      }
```

Each $\langle relation \rangle$ consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by `!`, and may not start with `?`. This evaluates to $+1$ if all comparisons $\langle operand_i \rangle \langle relation_i \rangle \langle operand_{i+1} \rangle$ are true, and $+0$ otherwise. All $\langle operands \rangle$ are evaluated (once) in all cases. See `\fp_compare:nTF` for details.

```
+ \fp_eval:n { <operand1> + <operand2> }
- \fp_eval:n { <operand1> - <operand2> }
```

Computes the sum or the difference of its two $\langle operands \rangle$. The “invalid operation” exception occurs for $\infty - \infty$. “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is NaN.

```

* \fp_eval:n { <operand1> * <operand2> }
/ \fp_eval:n { <operand1> / <operand2> }

```

Computes the product or the ratio of its two $\langle \text{operands} \rangle$. The “invalid operation” exception occurs for ∞/∞ , $0/0$, or $0 * \infty$. “Division by zero” occurs when dividing a finite non-zero number by ± 0 . “Underflow” and “overflow” occur when appropriate. When $\langle \text{operand}_1 \rangle$ is a tuple and $\langle \text{operand}_2 \rangle$ is a floating point number, each item of $\langle \text{operand}_1 \rangle$ is multiplied or divided by $\langle \text{operand}_2 \rangle$. Multiplication also supports the case where $\langle \text{operand}_1 \rangle$ is a floating point number and $\langle \text{operand}_2 \rangle$ a tuple. Other combinations yield an “invalid operation” exception and a NaN result.

```

+ \fp_eval:n { + <operand> }
- \fp_eval:n { - <operand> }
! \fp_eval:n { ! <operand> }

```

The unary $+$ does nothing, the unary $-$ changes the sign of the $\langle \text{operand} \rangle$ (for a tuple, of all its components), and $!$ $\langle \text{operand} \rangle$ evaluates to 1 if $\langle \text{operand} \rangle$ is false (is ± 0) and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.

```

** \fp_eval:n { <operand1> ** <operand2> }
^ \fp_eval:n { <operand1> ^ <operand2> }

```

Raises $\langle \text{operand}_1 \rangle$ to the power $\langle \text{operand}_2 \rangle$. This operation is right associative, hence $2 ** 2 ** 3$ equals $2^{2^3} = 256$. If $\langle \text{operand}_1 \rangle$ is negative or -0 then: the result’s sign is $+$ if the $\langle \text{operand}_2 \rangle$ is infinite and $(-1)^p$ if the $\langle \text{operand}_2 \rangle$ is $p/5^q$ with p, q integers; the result is $+0$ if $\text{abs}(\langle \text{operand}_1 \rangle)^\langle \text{operand}_2 \rangle$ evaluates to zero; in other cases the “invalid operation” exception occurs because the sign cannot be determined. “Division by zero” occurs when raising ± 0 to a finite strictly negative power. “Underflow” and “overflow” occur when appropriate. If either operand is a tuple, “invalid operation” occurs.

```

abs \fp_eval:n { abs( <fpexpr> ) }

```

Computes the absolute value of the $\langle \text{fpexpr} \rangle$. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also `\fp_abs:n`.

```

exp \fp_eval:n { exp( <fpexpr> ) }

```

Computes the exponential of the $\langle \text{fpexpr} \rangle$. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```

ln \fp_eval:n { ln( <fpexpr> ) }

```

Computes the natural logarithm of the $\langle \text{fpexpr} \rangle$. Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for $\ln(-0)$. “Division by zero” occurs when evaluating $\ln(+0) = -\infty$. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

```

max \fp_eval:n { max( <fpexpr1> , <fpexpr2> , ... ) }
min \fp_eval:n { min( <fpexpr1> , <fpexpr2> , ... ) }

```

Evaluates each $\langle \text{fpexpr} \rangle$ and computes the largest (smallest) of those. If any of the $\langle \text{fpexpr} \rangle$ is a NaN or tuple, the result is NaN. If any operand is a tuple, “invalid operation” occurs; these operations do not raise exceptions in other cases.

round	<code>\fp_eval:n { round (<fpexpr>) }</code>
trunc	<code>\fp_eval:n { round (<fpexpr₁> , <fpexpr₂>) }</code>
ceil	<code>\fp_eval:n { round (<fpexpr₁> , <fpexpr₂> , <fpexpr₃>) }</code>
floor	

New: 2013-12-14
Updated: 2015-08-08

Only **round** accepts a third argument. Evaluates $\langle fpexpr_1 \rangle = x$ and $\langle fpexpr_2 \rangle = n$ and $\langle fpexpr_3 \rangle = t$ then rounds x to n places. If n is an integer, this rounds x to a multiple of 10^{-n} ; if $n = +\infty$, this always yields x ; if $n = -\infty$, this yields one of ± 0 , $\pm\infty$, or NaN; if n is neither $\pm\infty$ nor an integer, then an “invalid operation” exception is raised. When $\langle fpexpr_2 \rangle$ is omitted, $n = 0$, *i.e.*, $\langle fpexpr_1 \rangle$ is rounded to an integer. The rounding direction depends on the function.

- **round** yields the multiple of 10^{-n} closest to x , with ties (x half-way between two such multiples) rounded as follows. If t is **nan** or not given the even multiple is chosen (“ties to even”), if $t = \pm 0$ the multiple closest to 0 is chosen (“ties to zero”), if t is positive/negative the multiple closest to $\infty/-\infty$ is chosen (“ties towards positive/negative infinity”).
- **floor** yields the largest multiple of 10^{-n} smaller or equal to x (“round towards negative infinity”);
- **ceil** yields the smallest multiple of 10^{-n} greater or equal to x (“round towards positive infinity”);
- **trunc** yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less than that of x (“round towards zero”).

“Overflow” occurs if x is finite and the result is infinite (this can only happen if $\langle fpexpr_2 \rangle < -9984$). If any operand is a tuple, “invalid operation” occurs.

sign	<code>\fp_eval:n { sign(<fpexpr>) }</code>
-------------	--

Evaluates the $\langle fpexpr \rangle$ and determines its sign: +1 for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0 , and NaN for NaN. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

sin	<code>\fp_eval:n { sin(<fpexpr>) }</code>
cos	<code>\fp_eval:n { cos(<fpexpr>) }</code>
tan	<code>\fp_eval:n { tan(<fpexpr>) }</code>
cot	<code>\fp_eval:n { cot(<fpexpr>) }</code>
csc	<code>\fp_eval:n { csc(<fpexpr>) }</code>
sec	<code>\fp_eval:n { sec(<fpexpr>) }</code>

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in radians. For arguments given in degrees, see **sind**, **cosd**, *etc.* Note that since π is irrational, $\sin(8\pi)$ is not quite zero, while its analogue $\text{sind}(8 \times 180)$ is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<code>sind</code>	<code>\fp_eval:n { sind(<fpexpr>) }</code>
<code>cosd</code>	<code>\fp_eval:n { cosd(<fpexpr>) }</code>
<code>tand</code>	<code>\fp_eval:n { tand(<fpexpr>) }</code>
<code>cotd</code>	<code>\fp_eval:n { cotd(<fpexpr>) }</code>
<code>cscd</code>	<code>\fp_eval:n { cscd(<fpexpr>) }</code>
<code>secd</code>	<code>\fp_eval:n { secd(<fpexpr>) }</code>

New: 2013-11-02

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$ given in degrees. For arguments given in radians, see `sin`, `cos`, *etc.* Note that since π is irrational, `sin(8pi)` is not quite zero, while its analogue `sind(8×180)` is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<code>asin</code>	<code>\fp_eval:n { asin(<fpexpr>) }</code>
<code>acos</code>	<code>\fp_eval:n { acos(<fpexpr>) }</code>
<code>acsc</code>	<code>\fp_eval:n { acsc(<fpexpr>) }</code>
<code>asec</code>	<code>\fp_eval:n { asec(<fpexpr>) }</code>

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fpexpr \rangle$ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for `asin` and `acsc` and $[0, \pi]$ for `acos` and `asec`. For a result in degrees, use `asind`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

<code>asind</code>	<code>\fp_eval:n { asind(<fpexpr>) }</code>
<code>acosd</code>	<code>\fp_eval:n { acosd(<fpexpr>) }</code>
<code>acscd</code>	<code>\fp_eval:n { acscd(<fpexpr>) }</code>
<code>asecd</code>	<code>\fp_eval:n { asecd(<fpexpr>) }</code>

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the $\langle fpexpr \rangle$ and returns the result in degrees, in the range $[-90, 90]$ for `asin` and `acsc` and $[0, 180]$ for `acos` and `asec`. For a result in radians, use `asin`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

atan	<code>\fp_eval:n { atan(<fpexpr>) }</code>
acot	<code>\fp_eval:n { atan(<fpexpr₁> , <fpexpr₂>) }</code>
<hr/>	
New: 2013-11-02	<code>\fp_eval:n { acot(<fpexpr>) }</code>
	<code>\fp_eval:n { acot(<fpexpr₁> , <fpexpr₂>) }</code>

Those functions yield an angle in radians: **atand** and **acotd** are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by π . Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm\pi/4, \pm 3\pi/4\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

atand	<code>\fp_eval:n { atand(<fpexpr>) }</code>
acotd	<code>\fp_eval:n { atand(<fpexpr₁> , <fpexpr₂>) }</code>
<hr/>	
New: 2013-11-02	<code>\fp_eval:n { acotd(<fpexpr>) }</code>
	<code>\fp_eval:n { acotd(<fpexpr₁> , <fpexpr₂>) }</code>

Those functions yield an angle in degrees: **atand** and **acotd** are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the $\langle fpexpr \rangle$: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$: this is the arctangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180 depending on the signs of $\langle fpexpr_1 \rangle$ and $\langle fpexpr_2 \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$, equal to the arccotangent of $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ need not be defined for the two-argument arctangent: when both expressions yield ± 0 , or when both yield $\pm \infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

sqrt	<code>\fp_eval:n { sqrt(<fpexpr>) }</code>
-------------	--

New: 2013-12-14 Computes the square root of the $\langle fpexpr \rangle$. The “invalid operation” is raised when the $\langle fpexpr \rangle$ is negative or is a tuple; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\text{NaN}} = \text{NaN}$.

<hr/> rand <hr/>	<code>\fp_eval:n { rand() }</code>
<hr/> New: 2016-12-05 <hr/>	Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. This is not yet available in X _Y TeX. The random seed can be queried using <code>\sys_rand_seed:</code> and set using <code>\sys_gset_rand_seed:n</code> .

TeXhackers note: This is based on pseudo-random numbers provided by the engine’s primitive `\pdfuniformdeviate` in pdfTeX, pTeX, upTeX and `\uniformdeviate` in LuaTeX. The underlying code is based on Metapost, which follows an additive scheme recommended in Section 3.6 of “The Art of Computer Programming, Volume 2”.

While we are more careful than `\uniformdeviate` to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.

<hr/> randint <hr/>	<code>\fp_eval:n { randint(<fpexpr>) }</code>
<hr/> New: 2016-12-05 <hr/>	<code>\fp_eval:n { randint(<fpexpr₁₂</code>
	Produces a pseudo-random integer between 1 and <code><fpexpr></code> or between <code><fpexpr_{1 and <code><fpexpr_{2 inclusive. The bounds must be integers in the range $(-10^{16}, 10^{16})$ and the first must be smaller or equal to the second. See rand for important comments on how these pseudo-random numbers are generated.}</code>}</code>

<hr/> inf nan <hr/>	The special values $+\infty$, $-\infty$, and NaN are represented as <code>inf</code> , <code>-inf</code> and <code>nan</code> (see <code>\c_minus_inf_fp</code> , <code>\c_minus_inf_fp</code> and <code>\c_nan_fp</code>).
--------------------------------------	--

<hr/> pi <hr/>	The value of π (see <code>\c_pi_fp</code>).
-----------------------	--

<hr/> deg <hr/>	The value of 1° in radians (see <code>\c_one_degree_fp</code>).
------------------------	---

<hr/> em ex in pt pc cm mm dd cc nd nc bp sp <hr/>	Those units of measurement are equal to their values in pt , namely <div style="margin-left: 100px;"> $1\text{in} = 72.27\text{pt}$ $1\text{pt} = 1\text{pt}$ $1\text{pc} = 12\text{pt}$ $1\text{cm} = \frac{1}{2.54}\text{in} = 28.45275590551181\text{pt}$ $1\text{mm} = \frac{1}{25.4}\text{in} = 2.845275590551181\text{pt}$ $1\text{dd} = 0.376065\text{mm} = 1.07000856496063\text{pt}$ $1\text{cc} = 12\text{dd} = 12.84010277952756\text{pt}$ $1\text{nd} = 0.375\text{mm} = 1.066978346456693\text{pt}$ $1\text{nc} = 12\text{nd} = 12.80374015748031\text{pt}$ $1\text{bp} = \frac{1}{72}\text{in} = 1.00375\text{pt}$ $1\text{sp} = 2^{-16}\text{pt} = 1.52587890625e - 5\text{pt}.$ </div>
---	--

The values of the (font-dependent) units **em** and **ex** are gathered from TeX when the surrounding floating point expression is evaluated.

<hr/> true false <hr/>	Other names for 1 and +0.
<hr/> \fp_abs:n ★ <small>New: 2012-05-14 Updated: 2012-07-08</small> <hr/>	\fp_abs:n { <i>floating point expression</i> } Evaluates the <i>floating point expression</i> as described for \fp_eval:n and leaves the absolute value of the result in the input stream. If the argument is a tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating point expressions, abs() can be used.
<hr/> \fp_max:nn ★ \fp_min:nn ★ <small>New: 2012-09-26</small> <hr/>	\fp_max:nn { <i>fp expression 1</i> } { <i>fp expression 2</i> } Evaluates the <i>floating point expressions</i> as described for \fp_eval:n and leaves the resulting larger (max) or smaller (min) value in the input stream. If the argument is a tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating point expressions, max() and min() can be used.

10 Disclaimer and roadmap

The package may break down if the escape character is among 0123456789_+, or if it receives a \TeX primitive conditional affected by **\exp_not:N**.

The following need to be done. I'll try to time-order the items.

- Function to count items in a tuple (and to determine if something is a tuple).
- Decide what exponent range to consider.
- Support signalling **nan**.
- Modulo and remainder, and rounding function **quantize** (and its friends analogous to **trunc**, **ceil**, **floor**).
- **\fp_format:nn** {*fpexpr*} {*format*}, but what should *format* be? More general pretty printing?
- Add **and**, **or**, **xor**? Perhaps under the names **all**, **any**, and **xor**?
- Add $\log(x, b)$ for logarithm of x in base b .
- **hypot** (Euclidean length). Cartesian-to-polar transform.
- Hyperbolic functions **cosh**, **sinh**, **tanh**.
- Inverse hyperbolics.
- Base conversion, input such as **0xAB.CDEF**.
- Factorial (not with **!**), gamma function.
- Improve coefficients of the **sin** and **tan** series.
- Treat upper and lower case letters identically in identifiers, and ignore underscores.
- Add an **array(1,2,3)** and **i=complex(0,1)**.

- Provide an experimental `map` function? Perhaps easier to implement if it is a single character, `@sin(1,2)?`
- Provide `\fp_if_nan:nTF`, and an `isnan` function?
- Support keyword arguments?

`Pgfmath` also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs, and tests to add.

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- Logarithms of numbers very close to 1 are inaccurate.
- When rounding towards $-\infty$, `\dim_to_fp:n {0pt}` should return -0 , not $+0$.
- The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.
- `0e9999999999` gives a \TeX “number too large” error.
- Subnormals are not implemented.

Possible optimizations/improvements.

- Document that `l3trial/l3fp-types` introduces tools for adding new types.
- In subsection 9.1, write a grammar.
- It would be nice if the `parse` auxiliaries for each operation were set up in the corresponding module, rather than centralizing in `l3fp-parse`.
- Some functions should get an `_o` ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the `ternary` set of functions is ugly.
- There are many `~` missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/(\lfloor 200x \rfloor + 1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of t . However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (`l3fp-basics`).
- Understand and document `__fp_basics_pack_weird_low:NNNNw` and `__fp_basics_pack_weird_high:NNNNNNNNw` better. Move the other `basics_pack` auxiliaries to `l3fp-aux` under a better name.

- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan's articles, some previous T_EX fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, @/ or whatever)?

Part XXII

The l3intarray package: fast global integer arrays

1 l3intarray documentation

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast l3seq, where access time is linear). These arrays have several important features

- The size of the array is fixed and must be given at point of initialisation
- The absolute value of each entry has maximum $2^{30} - 1$ (*i.e.* one power lower than the usual `\c_max_int` ceiling of $2^{31} - 1$)

The use of `intarray` data is therefore recommended for cases where the need for fast access is of paramount importance.

<hr/> <code>\intarray_new:Nn</code> <hr/>	<code>\intarray_new:Nn <intarray var> {<size>}</code>
<hr/> New: 2018-03-29 <hr/>	Evaluates the integer expression <code><size></code> and allocates an <i><integer array variable></i> with that number of (zero) entries.
<hr/> <code>\intarray_count:N</code> ★ <hr/>	<code>\intarray_count:N <intarray var></code>
<hr/> New: 2018-03-29 <hr/>	Expands to the number of entries in the <i><integer array variable></i> . Contrarily to <code>\seq_count:N</code> this is performed in constant time.
<hr/> <code>\intarray_gset:Nnn</code> <hr/>	<code>\intarray_gset:Nnn <intarray var> {<position>} {<value>}</code>
<hr/> New: 2018-03-29 <hr/>	Stores the result of evaluating the integer expression <code><value></code> into the <i><integer array variable></i> at the (integer expression) <code><position></code> . If the <code><position></code> is not between 1 and the <code>\intarray_count:N</code> , or the <code><value></code> 's absolute value is bigger than $2^{30} - 1$, an error occurs. Assignments are always global.
<hr/> <code>\intarray_item:Nn</code> ★ <hr/>	<code>\intarray_item:Nn <intarray var> {<position>}</code>
<hr/> New: 2018-03-29 <hr/>	Expands to the integer entry stored at the (integer expression) <code><position></code> in the <i><integer array variable></i> . If the <code><position></code> is not between 1 and the <code>\intarray_count:N</code> , an error occurs.

1.1 Implementation notes

It is a wrapper around the `\fontdimen` primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30} - 1$). In contrast to `l3seq` sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive `\fontdimen` stores dimensions but the `l3intarray` package transparently converts these from/to integers. Assignments are always global.

While LuaTeX's memory is extensible, other engines can “only” deal with a bit less than 4×10^6 entries in all `\fontdimen` arrays combined (with default TeXLive settings).

Part XXIII

The l3sort package

Sorting functions

1 Controlling sorting

L^AT_EX3 comes with a facility to sort list variables (sequences, token lists, or comma-lists) according to some user-defined comparison. For instance,

```
\clist_set:Nn \l_foo_clist { 3 , 01 , -2 , 5 , +1 }
\clist_sort:Nn \l_foo_clist
{
  \int_compare:nNnTF { #1 } > { #2 }
  { \sort_return_swapped: }
  { \sort_return_same: }
}
```

results in `\l_foo_clist` holding the values `{ -2 , 01 , +1 , 3 , 5 }` sorted in non-decreasing order.

The code defining the comparison should call `\sort_return_swapped:` if the two items given as `#1` and `#2` are not in the correct order, and otherwise it should call `\sort_return_same:` to indicate that the order of this pair of items should not be changed.

For instance, a *comparison code* consisting only of `\sort_return_same:` with no test yields a trivial sort: the final order is identical to the original order. Conversely, using a *comparison code* consisting only of `\sort_return_swapped:` reverses the list (in a fairly inefficient way).

T_EXhackers note: The current implementation is limited to sorting approximately 20000 items (40000 in LuaT_EX), depending on what other packages are loaded.

Internally, the code from l3sort stores items in `\toks` registers allocated locally. Thus, the *comparison code* should not call `\newtoks` or other commands that allocate new `\toks` registers. On the other hand, altering the value of a previously allocated `\toks` register is not a problem.

```
\sort_return_same:
\sort_return_swapped:
```

New: 2017-02-06

```
\seq_sort:Nn <seq var>
{ ... \sort_return_same: or \sort_return_swapped: ... }
```

Indicates whether to keep the order or swap the order of two items that are compared in the sorting code. Only one of the `\sort_return_...` functions should be used by the code, according to the results of some tests on the items `#1` and `#2` to be compared.

Part XXIV

The l3tl-analysis package: Analysing token lists

1 l3tl-analysis documentation

This module mostly provides internal functions for use in the l3regex module. However, it provides as a side-effect a user debugging function, very similar to the \ShowTokens macro from the ted package.

\tl_analysis_show:N	\tl_analysis_show:n {\token list}
\tl_analysis_show:n	

New: 2018-04-09

Displays to the terminal the detailed decomposition of the $\langle token list \rangle$ into tokens, showing the category code of each character token, the meaning of control sequences and active characters, and the value of registers.

\tl_analysis_map_inline:nn	\tl_analysis_map_inline:nn {\token list} {\inline function}
\tl_analysis_map_inline:Nn	

New: 2018-04-09

Applies the $\langle inline function \rangle$ to each individual $\langle token \rangle$ in the $\langle token list \rangle$. The $\langle inline function \rangle$ receives three arguments:

- $\langle tokens \rangle$, which both o-expand and x-expand to the $\langle token \rangle$. The detailed form of $\langle token \rangle$ may change in later releases.
- $\langle char code \rangle$, a decimal representation of the character code of the token, -1 if it is a control sequence (with $\langle catcode \rangle 0$).
- $\langle catcode \rangle$, a capital hexadecimal digit which denotes the category code of the $\langle token \rangle$ (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C:other, D:active).

Part XXV

The `l3regex` package: Regular expressions in `TEX`

The `l3regex` package provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX), with some additions due to the fact that `TEX` manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

```
\tl_set:Nn \l_my_tl { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
```

the token list variable `\l_my_tl` holds the text “`This cat.`”, where the first occurrence of “`at`” was replaced by “`is`”. A more complicated example is a pattern to emphasize each word and add a comma after it:

```
\regex_replace_all:nnN { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
```

The `\w` sequence represents any “word” character, and `+` indicates that the `\w` sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, `\0` denotes the full match (here, a word). The command `\emph` is inserted using `\c{emph}`, and its argument `\0` is put between braces `\cB\{` and `\cE\}`.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using `\regex_const:Nn`. For example,

```
\regex_const:Nn \c_foo_regex { \c{begin} \cB. (\c[~BE].*) \cE. }
```

stores in `\c_foo_regex` a regular expression which matches the starting marker for an environment: `\begin`, followed by a begin-group token (`\cB.`), then any number of tokens which are neither begin-group nor end-group character tokens (`\c[~BE].*`), ending with an end-group token (`\cE.`). As explained in the next section, the parentheses “capture” the result of `\c[~BE].*`, giving us access to the name of the environment when doing replacements.

1 Syntax of regular expressions

We start with a few examples, and encourage the reader to apply `\regex_show:n` to these regular expressions.

- `Cat` matches the word “Cat” capitalized in this way, but also matches the beginning of the word “Cattle”: use `\bCat\b` to match a complete word only.
- `[abc]` matches one letter among “a”, “b”, “c”; the pattern `(a|b|c)` matches the same three possible letters (but see the discussion of submatches below).
- `[A-Za-z]*` matches any number (due to the quantifier `*`) of Latin letters (not accented).

- `\c{[A-Za-z]*}` matches a control sequence made of Latin letters.
- `_[^\\]*_` matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to `_.*?_` where `.` matches arbitrary characters and the lazy quantifier `*?` means to match as few characters as possible, thus avoiding matching underscores.
- `[+-]?\d+` matches an explicit integer with at most one sign.
- `[+\\-_]*\d+_*` matches an explicit integer with any number of `+` and `-` signs, with spaces allowed except within the mantissa, and surrounded by spaces.
- `[+\\-_]*(\d+|\\d*\\.\\d+)_*` matches an explicit integer or decimal number; using `[.,]` instead of `\\.` would allow the comma as a decimal marker.
- `[+\\-_]*(\d+|\\d*\\.\\d+)_*(\\(?i\\)pt|in|[cem]m|ex|[bs]p|[dn]d|[pcn]c)_*` matches an explicit dimension with any unit that T_EX knows, where `(?i)` means to treat lowercase and uppercase letters identically.
- `[+\\-_]*(\\(?i\\)nan|inf|(\d+|\\d*\\.\\d+)(_.*e[+-_]*\d+)?)_*` matches an explicit floating point number or the special values `nan` and `inf` (with signs and spaces allowed).
- `[+\\-_]*(\d+|\\cC.)_*` matches an explicit integer or control sequence (without checking whether it is an integer variable).
- `\\G.*?\\K` at the beginning of a regular expression matches and discards (due to `\\K`) everything between the end of the previous match (`\\G`) and what is matched by the rest of the regular expression; this is useful in `\\regex_replace_all:nnN` when the goal is to extract matches or submatches in a finer way than with `\\regex_extract_all:nnN`.

While it is impossible for a regular expression to match only integer expressions, `[+\\-\\(\\)*\\d+\\)([+\\-*/][+\\-\\(\\)*\\d+\\)]*` matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.

Most characters match exactly themselves, with an arbitrary category code. Some characters are special and must be escaped with a backslash (*e.g.*, `*` matches a star character). Some escape sequences of the form backslash-letter also have a special meaning (for instance `\\d` matches any digit). As a rule,

- every alphanumeric character (`A-Z`, `a-z`, `0-9`) matches exactly itself, and should not be escaped, because `\\A`, `\\B`, ... have special meanings;
- non-alphanumeric printable ascii characters can (and should) always be escaped: many of them have special meanings (*e.g.*, use `\\(`, `\\)`, `\\?`, `\\.`);
- spaces should always be escaped (even in character classes);
- any other character may be escaped or not, without any effect: both versions match exactly that character.

Note that these rules play nicely with the fact that many non-alphanumeric characters are difficult to input into T_EX under normal category codes. For instance, `\abc%` matches the characters `\abc%` (with arbitrary category codes), but does not match the control sequence `\abc` followed by a percent character. Matching control sequences can be done using the `\c{<regex>}` syntax (see below).

Any special character which appears at a place where its special behaviour cannot apply matches itself instead (for instance, a quantifier appearing at the beginning of a string), after raising a warning.

Characters.

`\x{hh...}` Character with hex code `hh...`

`\xhh` Character with hex code `hh`.

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

`\f` Form-feed (hex 0C).

`\n` New line (hex 0A).

`\r` Carriage return (hex 0D).

`\t` Horizontal tab (hex 09).

Character types.

`.` A single period matches any token.

`\d` Any decimal digit.

`\h` Any horizontal space character, equivalent to `[\ \^^I]`: space and tab.

`\s` Any space character, equivalent to `[\ \^^I\^^J\^^L\^^M]`.

`\v` Any vertical space character, equivalent to `[\^^J\^^K\^^L\^^M]`. Note that `\^^K` is a vertical space, but not a space, for compatibility with Perl.

`\w` Any word character, *i.e.*, alphanumerics and underscore, equivalent to the explicit class `[A-Za-z0-9_]`.

`\D` Any token not matched by `\d`.

`\H` Any token not matched by `\h`.

`\N` Any token other than the `\n` character (hex 0A).

`\S` Any token not matched by `\s`.

`\V` Any token not matched by `\v`.

`\W` Any token not matched by `\w`.

Of those, `.`, `\D`, `\H`, `\N`, `\S`, `\V`, and `\W` match arbitrary control sequences.

Character classes match exactly one token in the subject.

`[...]` Positive character class. Matches any of the specified tokens.

[**^...**] Negative character class. Matches any token other than the specified characters.

x-y Within a character class, this denotes a range (can be used with escaped characters).

[:**<name>**:] Within a character class (one more set of brackets), this denotes the POSIX character class **<name>**, which can be **alnum**, **alpha**, **ascii**, **blank**, **cntrl**, **digit**, **graph**, **lower**, **print**, **punct**, **space**, **upper**, **word**, or **xdigit**.

[:**~<name>**:] Negative POSIX character class.

For instance, [**a-oq-z\cC.**] matches any lowercase latin letter except **p**, as well as control sequences (see below for a description of **\c**).

Quantifiers (repetition).

? 0 or 1, greedy.

?? 0 or 1, lazy.

***** 0 or more, greedy.

***?** 0 or more, lazy.

+ 1 or more, greedy.

+? 1 or more, lazy.

{n} Exactly *n*.

{n,} *n* or more, greedy.

{n,}? *n* or more, lazy.

{n,m} At least *n*, no more than *m*, greedy.

{n,m}? At least *n*, no more than *m*, lazy.

Anchors and simple assertions.

\b Word boundary: either the previous token is matched by **\w** and the next by **\W**, or the opposite. For this purpose, the ends of the token list are considered as **\W**.

\B Not a word boundary: between two **\w** tokens or two **\W** tokens (including the boundary).

^ or **\A** Start of the subject token list.

\$, **\Z** or **\z** End of the subject token list.

\G Start of the current match. This is only different from **^** in the case of multiple matches: for instance **\regex_count:nnN { \G a } { aaba } \l_tmpa_int** yields 2, but replacing **\G** by **^** would result in **\l_tmpa_int** holding the value 1.

Alternation and capturing groups.

A|B|C Either one of **A**, **B**, or **C**.

(...) Capturing group.

(?:...) Non-capturing group.

(?<|...) Non-capturing group which resets the group number for capturing groups in each alternative. The following group is numbered with the first unused group number.

The `\c` escape sequence allows to test the category code of tokens, and match control sequences. Each character category is represented by a single uppercase letter:

- C for control sequences;
- B for begin-group tokens;
- E for end-group tokens;
- M for math shift;
- T for alignment tab tokens;
- P for macro parameter tokens;
- U for superscript tokens (up);
- D for subscript tokens (down);
- S for spaces;
- L for letters;
- O for others; and
- A for active characters.

The `\c` escape sequence is used as follows.

`\c{<regex>}` A control sequence whose *cname* matches the *<regex>*, anchored at the beginning and end, so that `\c{begin}` matches exactly `\begin`, and nothing else.

`\cX` Applies to the next object, which can be a character, character property, class, or group, and forces this object to only match tokens with category **X** (any of CBEMTPUDSLOA). For instance, `\cL[A-Z\d]` matches uppercase letters and digits of category code letter, `\cC.` matches any control sequence, and `\cO(abc)` matches `abc` where each character has category other.

`\c[XYZ]` Applies to the next object, and forces it to only match tokens with category **X**, **Y**, or **Z** (each being any of CBEMTPUDSLOA). For instance, `\c[LSO](..)` matches two tokens of category letter, space, or other.

`\c[^XYZ]` Applies to the next object and prevents it from matching any token with category **X**, **Y**, or **Z** (each being any of CBEMTPUDSLOA). For instance, `\c[^O]\d` matches digits which have any category different from other.

The category code tests can be used inside classes; for instance, `[\cO\d \c[LO][A-F]]` matches what \TeX considers as hexadecimal digits, namely digits with category other, or uppercase letters from **A** to **F** with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, `\cL(ab\cO*cd)` matches `ab*cd` where all characters are of category letter, except `*` which has category other.

The `\u` escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters.

Namely, `\u{<tl var name>}` matches the exact contents of the token list `<tl var>`. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are not supported directly: use a group.

The option `(?i)` makes the match case insensitive (identifying A–Z with a–z; no Unicode support yet). This applies until the end of the group in which it appears, and can be reverted using `(?-i)`. For instance, in `(?i)(a(?-i)b|c)d`, the letters `a` and `d` are affected by the `i` option. Characters within ranges and classes are affected individually: `(?i)[Y-\]` is equivalent to `[YZ\[\]yz]`, and `(?i)[^aeiou]` matches any character which is not a vowel. Neither character properties, nor `\c{...}` nor `\u{...}` are affected by the `i` option.

In character classes, only `[`, `^`, `-`, `]`, `\` and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. Any escape sequence which matches a single character (`\d`, `\D`, *etc.*) is supported in character classes. If the first character is `^`, then the meaning of the character class is inverted; `^` appearing anywhere else in the range is not special. If the first character (possibly following a leading `^`) is `]` then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using `-`, for instance, `[\D 0–5]` and `[\^6–9]` are equivalent.

Capturing groups are a means of extracting information about the match. Parenthesized groups are labelled in the order of their opening parenthesis, starting at 1. The contents of those groups corresponding to the “best” match (leftmost longest) can be extracted and stored in a sequence of token lists using for instance `\regex_extract_once:nnTF`.

The `\K` escape sequence resets the beginning of the match to the current position in the token list. This only affects what is reported as the full match. For instance,

```
\regex_extract_all:nnN { a \K . } { a123aaxyz } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{1}` and `{a}`: the true matches are `{a1}` and `{aa}`, but they are trimmed by the use of `\K`. The `\K` command does not affect capturing groups: for instance,

```
\regex_extract_once:nnN { (. \K c)+ \d } { acbc3 } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{c3}` and `{bc}`: the true match is `{acbc3}`, with first submatch `{bc}`, but `\K` resets the beginning of the match to the last position where it appears.

2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Backslash introduces various special constructions, described further below:

- `\0` is the whole match;
- `\1` is the submatch that was matched by the first (capturing) group `(...)`; similarly for `\2`, ..., `\9` and `\g{<number>}`;
- `_` inserts a space (spaces are ignored when not escaped);

- `\a, \e, \f, \n, \r, \t, \xhh, \x{hhh}` correspond to single characters as in regular expressions;
- `\c{<cs name>}` inserts a control sequence;
- `\c{<category>}<character>` (see below);
- `\u{<tl var name>}` inserts the contents of the `<tl var>` (see below).

Characters other than backslash and space are simply inserted in the result (but since the replacement text is first converted to a string, one should also escape characters that are special for T_EX, for instance use `\#`). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

```
\tl_set:Nn \l_my_tl { Hello,~world! }
\regex_replace_all:nnN { ([er]?l|o) . } { (\0--\1) } \l_my_tl
```

results in `\l_my_tl` holding `H(e1l--e1)(o,--o) w(or--o)(ld--l)!`

The submatches are numbered according to the order in which the opening parenthesis of capturing groups appear in the regular expression to match. The n -th submatch is empty if there are fewer than n capturing groups or for capturing groups that appear in alternatives that were not used for the match. In case a capturing group matches several times during a match (due to quantifiers) only the last match is used in the replacement text. Submatches always keep the same category codes as in the original token list.

The characters inserted by the replacement have category code 12 (other) by default, with the exception of space characters. Spaces inserted through `_` have category code 10, while spaces inserted through `\x20` or `\x{20}` have category code 12. The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\cX(...)` Produces the characters “...” with category `X`, which must be one of `CBEMTPUDSLOA` as in regular expressions. Parentheses are optional for a single character (which can be an escape sequence). When nested, the innermost category code applies, for instance `\cL(Hello\cS\ world)!` gives this text with standard category codes.

`\c{<text>}` Produces the control sequence with csname `<text>`. The `<text>` may contain references to the submatches `\0`, `\1`, and so on, as in the example for `\u` below.

The escape sequence `\u{<tl var name>}` allows to insert the contents of the token list with name `<tl var name>` directly into the replacement, giving an easier control of category codes. When nested in `\c{...}` and `\u{...}` constructions, the `\u` and `\c` escape sequences perform `\tl_to_str:v`, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of `\c` and `\u`. For instance,

```
\tl_set:Nn \l_my_one_tl { first }
\tl_set:Nn \l_my_two_tl { \emph{second} }
\tl_set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { [,]+ } { \u{1_my_\0_tl} } \l_my_tl
```

results in `\l_my_tl` holding `first,\emph{second},first,first`.

3 Pre-compiling regular expressions

If a regular expression is to be used several times, it is better to compile it once rather than doing it each time the regular expression is used. The compiled regular expression is stored in a variable. All of the `l3regex` module's functions can be given their regular expression argument either as an explicit string or as a compiled regular expression.

`\regex_new:N`

New: 2017-05-26

`\regex_new:N` $\langle regex\ var \rangle$

Creates a new $\langle regex\ var \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle regex\ var \rangle$ is initially such that it never matches.

`\regex_set:Nn`
`\regex_gset:Nn`
`\regex_const:Nn`

New: 2017-05-26

`\regex_set:Nn` $\langle regex\ var \rangle$ $\{ \langle regex \rangle \}$

Stores a compiled version of the $\langle regular\ expression \rangle$ in the $\langle regex\ var \rangle$. For instance, this function can be used as

```
\regex_new:N \l_my_regex
\regex_set:Nn \l_my_regex { my\ (simple\ )? reg(ex|ular\ expression) }
```

The assignment is local for `\regex_set:Nn` and global for `\regex_gset:Nn`. Use `\regex_const:Nn` for compiled expressions which never change.

`\regex_show:n`
`\regex_show:N`

New: 2017-05-26

`\regex_show:n` $\{ \langle regex \rangle \}$

Shows how `l3regex` interprets the $\langle regex \rangle$. For instance, `\regex_show:n { \A X|Y }` shows

```
+--branch
  anchor at start (\A)
  char code 88
+--branch
  char code 89
```

indicating that the anchor `\A` only applies to the first branch: the second branch is not anchored to the beginning of the match.

4 Matching

All regular expression functions are available in both `:n` and `:N` variants. The former require a “standard” regular expression, while the later require a compiled expression as generated by `\regex_(g)set:Nn`.

`\regex_match:nnTF`
`\regex_match:NnTF`

New: 2017-05-26

`\regex_match:nnTF` $\{ \langle regex \rangle \}$ $\{ \langle token\ list \rangle \}$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$

Tests whether the $\langle regular\ expression \rangle$ matches any part of the $\langle token\ list \rangle$. For instance,

```
\regex_match:nnTF { b [cde]* } { abedcdx } { TRUE } { FALSE }
\regex_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }
```

leaves `TRUE` then `FALSE` in the input stream.

```
\regex_count:nnN
\regex_count:NnN
```

New: 2017-05-26

```
\regex_count:nnN {<regex>} {<token list>} <int var>
```

Sets *<int var>* within the current T_EX group level equal to the number of times *<regular expression>* appears in *<token list>*. The search starts by finding the left-most longest match, respecting greedy and lazy (non-greedy) operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,

```
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbababcb } \l_foo_int
```

results in `\l_foo_int` taking the value 5.

5 Submatch extraction

```
\regex_extract_once:nnN
\regex_extract_once:nnNTF
\regex_extract_once:NnN
\regex_extract_once:NnNTF
```

New: 2017-05-26

```
\regex_extract_once:nnN {<regex>} {<token list>} <seq var>
\regex_extract_once:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}
```

Finds the first match of the *<regular expression>* in the *<token list>*. If it exists, the match is stored as the first item of the *<seq var>*, and further items are the contents of capturing groups, in the order of their opening parenthesis. The *<seq var>* is assigned locally. If there is no match, the *<seq var>* is cleared. The testing versions insert the *<true code>* into the input stream if a match was found, and the *<false code>* otherwise.

For instance, assume that you type

```
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
{ true } { false }
```

Then the regular expression (anchored at the start with `\A` and at the end with `\Z`) must match the whole token list. The first capturing group, `(La)?`, matches `La`, and the second capturing group, `(!*)`, matches `!!!`. Thus, `\l_foo_seq` contains as a result the items `{LaTeX!!!}`, `{La}`, and `{!!!}`, and the `true` branch is left in the input stream. Note that the n -th item of `\l_foo_seq`, as obtained using `\seq_item:Nn`, correspond to the submatch numbered $(n - 1)$ in functions such as `\regex_replace_once:nnN`.

```
\regex_extract_all:nnN
\regex_extract_all:nnNTF
\regex_extract_all:NnN
\regex_extract_all:NnNTF
```

New: 2017-05-26

```
\regex_extract_all:nnN {<regex>} {<token list>} <seq var>
\regex_extract_all:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}
```

Finds all matches of the *<regular expression>* in the *<token list>*, and stores all the submatch information in a single sequence (concatenating the results of multiple `\regex_extract_once:nnN` calls). The *<seq var>* is assigned locally. If there is no match, the *<seq var>* is cleared. The testing versions insert the *<true code>* into the input stream if a match was found, and the *<false code>* otherwise. For instance, assume that you type

```
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
{ true } { false }
```

Then the regular expression matches twice, the resulting sequence contains the two items `{Hello}` and `{world}`, and the `true` branch is left in the input stream.

```
\regex_split:nnN
\regex_split:nnNTF
\regex_split:NnN
\regex_split:NnNTF
```

New: 2017-05-26

```
\regex_split:nnN {<regular expression>} {<token list>} <seq var>
\regex_split:nnNTF {<regular expression>} {<token list>} <seq var> {<true code>}
{<false code>}
```

Splits the *<token list>* into a sequence of parts, delimited by matches of the *<regular expression>*. If the *<regular expression>* has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to *<seq var>* is local. If no match is found the resulting *<seq var>* has the *<token list>* as its sole item. If the *<regular expression>* matches the empty token list, then the *<token list>* is split into single tokens. The testing versions insert the *<true code>* into the input stream if a match was found, and the *<false code>* otherwise. For example, after

```
\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
{ true } { false }
```

the sequence `\l_path_seq` contains the items `{the}`, `{path}`, `{for}`, `{this}`, and `{file.tex}`, and the `true` branch is left in the input stream.

6 Replacement

```
\regex_replace_once:nnN
\regex_replace_once:nnNTF
\regex_replace_once:NnN
\regex_replace_once:NnNTF
```

New: 2017-05-26

```
\regex_replace_once:nnN {<regular expression>} {<replacement>} <tl var>
\regex_replace_once:nnNTF {<regular expression>} {<replacement>} <tl var> {<true
code>} {<false code>}
```

Searches for the *<regular expression>* in the *<token list>* and replaces the first match with the *<replacement>*. The result is assigned locally to *<tl var>*. In the *<replacement>*, `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.*

```
\regex_replace_all:nnN
\regex_replace_all:nnNTF
\regex_replace_all:NnN
\regex_replace_all:NnNTF
```

New: 2017-05-26

```
\regex_replace_all:nnN {<regular expression>} {<replacement>} <tl var>
\regex_replace_all:nnNTF {<regular expression>} {<replacement>} <tl var> {<true
code>} {<false code>}
```

Replaces all occurrences of the *<regular expression>* in the *<token list>* by the *<replacement>*, where `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.* Every match is treated independently, and matches cannot overlap. The result is assigned locally to *<tl var>*.

7 Constants and variables

```
\l_tmpa_regex
\l_tmpb_regex
```

New: 2017-12-11

Scratch regex for local assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

```
\g_tmpa_regex
\g_tmpb_regex
```

New: 2017-12-11

Scratch regex for global assignment. These are never used by the kernel code, and so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

8 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Rewrite the documentation in a more ordered way, perhaps add a BNF?

Additional error-checking to come.

- Clean up the use of messages.
- Cleaner error reporting in the replacement phase.
- Add tracing information.
- Detect attempts to use back-references and other non-implemented syntax.
- Test for the maximum register `\c_max_register_int`.
- Find out whether the fact that `\W` and friends match the end-marker leads to bugs. Possibly update `__regex_item_reverse:n`.
- The empty `cs` should be matched by `\c{}`, not by `\c{csname.?endcsname\s?}`.

Code improvements to come.

- Shift arrays so that the useful information starts at position 1.
- Only build `.,` once.
- Use arrays for the left and right state stacks when compiling a regex.
- Should `__regex_action_free_group:n` only be used for greedy `{n,}` quantifier? (I think not.)
- Quantifiers for `\u` and assertions.
- When matching, keep track of an explicit stack of `current_state` and `current_submatches`.
- If possible, when a state is reused by the same thread, kill other subthreads.
- Use an array rather than `\l__regex_balance_tl` to build the function `__regex_replacement_balance_one_match:n`.
- Reduce the number of epsilon-transitions in alternatives.
- Optimize simple strings: use less states (`abcade` should give two states, for `abc` and `ade`). [Does that really make sense?]
- Optimize groups with no alternative.
- Optimize states with a single `__regex_action_free:n`.
- Optimize the use of `__regex_action_success:` by inserting it in state 2 directly instead of having an extra transition.
- Optimize the use of `\int_step...` functions.

- Groups don't capture within regexes for csnames; optimize and document.
- Better “show” for anchors, properties, and catcode tests.
- Does \K really need a new state for itself?
- When compiling, use a boolean `in_cs` and less magic numbers.
- Instead of checking whether the character is special or alphanumeric using its character code, check if it is special in regexes with `\cs_if_exist` tests.

The following features are likely to be implemented at some point in the future.

- General look-ahead/behind assertions.
- Regex matching on external files.
- Conditional subpatterns with look ahead/behind: “if what follows is [...], then [...]”.
- `(*..)` and `(?..)` sequences to set some options.
- UTF-8 mode for pdfTeX.
- Newline conventions are not done. In particular, we should have an option for `.` not to match newlines. Also, `\A` should differ from `^`, and `\Z`, `\z` and `$` should differ.
- Unicode properties: `\p{..}` and `\P{..}`; `\X` which should match any “extended” Unicode sequence. This requires to manipulate a lot of data, probably using tree-boxes.
- Provide a syntax such as `\ur{1_my_regex}` to use an already-compiled regex in a more complicated regex. This makes regexes more easily composable.
- Allowing `\u{1_my_t1}` in more places, for instance as the number of repetitions in a quantifier.

The following features of PCRE or Perl may or may not be implemented.

- Callout with `(?C...)` or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential `\regex_break`: and then of playing well with `\t1_map_break`: called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since `\fontdimen` are global.
- Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn't it?
- Named subpatterns: TeX programmers have lived so far without any need for named macro parameters.

The following features of PCRE or Perl will definitely not be implemented.

- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.

- Recursion: this is a non-regular feature.
- Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.
- Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.
- Backtracking control verbs: intrinsically tied to backtracking.
- `\ddd`, matching the character with octal code `ddd`: we already have `\x{...}` and the syntax is confusingly close to what we could have used for backreferences (`\1`, `\2`, ...), making it harder to produce useful error message.
- `\cx`, similar to $\mathrm{T}_{\mathrm{E}}\mathrm{X}$'s own `\^^x`.
- Comments: $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ already has its own system for comments.
- `\Q...\E` escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- `\C` single byte in UTF-8 mode: $\mathrm{X}_{\mathrm{Y}}\mathrm{T}_{\mathrm{E}}\mathrm{X}$ and $\mathrm{LuaT}_{\mathrm{E}}\mathrm{X}$ serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

Part XXVI

The l3box package

Boxes

There are three kinds of box operations: horizontal mode denoted with prefix `\hbox_`, vertical mode with prefix `\vbox_`, and the generic operations working in both modes with prefix `\box_`.

1 Creating and initialising boxes

<code>\box_new:N</code>	<code>\box_new:N</code> $\langle box \rangle$
<code>\box_new:c</code>	Creates a new $\langle box \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle box \rangle$ is initially void.

<code>\box_clear:N</code>	<code>\box_clear:N</code> $\langle box \rangle$
<code>\box_clear:c</code>	Clears the content of the $\langle box \rangle$ by setting the box equal to <code>\c_empty_box</code> .
<code>\box_gclear:N</code>	
<code>\box_gclear:c</code>	

<code>\box_clear_new:N</code>	<code>\box_clear_new:N</code> $\langle box \rangle$
<code>\box_clear_new:c</code>	Ensures that the $\langle box \rangle$ exists globally by applying <code>\box_new:N</code> if necessary, then applies <code>\box_(g)clear:N</code> to leave the $\langle box \rangle$ empty.
<code>\box_gclear_new:N</code>	
<code>\box_gclear_new:c</code>	

<code>\box_set_eq:NN</code>	<code>\box_set_eq:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
<code>\box_set_eq:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$.
<code>\box_gset_eq:NN</code>	
<code>\box_gset_eq:(cN Nc cc)</code>	

<code>\box_set_eq_clear:NN</code>	<code>\box_set_eq_clear:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
<code>\box_set_eq_clear:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ within the current TeX group equal to that of $\langle box_2 \rangle$, then clears $\langle box_2 \rangle$ globally.

<code>\box_gset_eq_clear:NN</code>	<code>\box_gset_eq_clear:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
<code>\box_gset_eq_clear:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$, then clears $\langle box_2 \rangle$. These assignments are global.

<code>\box_if_exist_p:N</code> ★	<code>\box_if_exist_p:N</code> $\langle box \rangle$
<code>\box_if_exist_p:c</code> ★	<code>\box_if_exist:NTF</code> $\langle box \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$
<code>\box_if_exist:NTF</code> ★	Tests whether the $\langle box \rangle$ is currently defined. This does not check that the $\langle box \rangle$ really is a box.
<code>\box_if_exist:cTF</code> ★	

New: 2012-03-03

2 Using boxes

`\box_use:N`
`\box_use:c`

`\box_use:N` $\langle box \rangle$

Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting.

T_EXhackers note: This is the T_EX primitive `\copy`.

`\box_use_drop:N`
`\box_use_drop:c`

`\box_use_drop:N` $\langle box \rangle$

Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting. The $\langle box \rangle$ is then cleared at the group level the box was set at, *i.e.* the current content is “dropped” entirely. For example, with

```
\hbox_set:Nn \l_tmpa_box { A }
\group_begin:
  \hbox_set:Nn \l_tmpa_box { B }
  \group_begin:
    \box_use_drop:N \l_tmpa_box
  \group_end:
  \box_show:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
```

the first use of `\box_show:N` will show an entirely cleared (void) box, and the second will show the letter A in the box.

This function is useful as boxes can contain an open-ended amount of material. As such, they can have a significant memory impact on T_EX. At the same time, it is often the case that once a box has been inserted, it is no longer needed at all. Using `\box_use_drop:N` in these circumstances therefore offers improved memory use and performance. It should therefore be preferred over `\box_use:N` where it is clear that the content is no longer needed in the variable.

T_EXhackers note: This is the T_EX primitive `\box`.

`\box_move_right:nn`
`\box_move_left:nn`

`\box_move_right:nn` $\{\langle dimexpr \rangle\} \{\langle box function \rangle\}$

This function operates in vertical mode, and inserts the material specified by the $\langle box function \rangle$ such that its reference point is displaced horizontally by the given $\langle dimexpr \rangle$ from the reference point for typesetting, to the right or left as appropriate. The $\langle box function \rangle$ should be a box operation such as `\box_use:N` $\langle box \rangle$ or a “raw” box specification such as `\vbox:n { xyz }`.

`\box_move_up:nn`
`\box_move_down:nn`

`\box_move_up:nn` $\{\langle dimexpr \rangle\} \{\langle box function \rangle\}$

This function operates in horizontal mode, and inserts the material specified by the $\langle box function \rangle$ such that its reference point is displaced vertically by the given $\langle dimexpr \rangle$ from the reference point for typesetting, up or down as appropriate. The $\langle box function \rangle$ should be a box operation such as `\box_use:N` $\langle box \rangle$ or a “raw” box specification such as `\vbox:n { xyz }`.

3 Measuring and setting box dimensions

<code>\box_dp:N</code>	<code>\box_dp:N</code> $\langle box \rangle$
<code>\box_dp:c</code>	Calculates the depth (below the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

T_EXhackers note: This is the T_EX primitive `\dp`.

<code>\box_ht:N</code>	<code>\box_ht:N</code> $\langle box \rangle$
<code>\box_ht:c</code>	Calculates the height (above the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

T_EXhackers note: This is the T_EX primitive `\ht`.

<code>\box_wd:N</code>	<code>\box_wd:N</code> $\langle box \rangle$
<code>\box_wd:c</code>	Calculates the width of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

T_EXhackers note: This is the T_EX primitive `\wd`.

<code>\box_set_dp:Nn</code>	<code>\box_set_dp:Nn</code> $\langle box \rangle$ $\{\langle dimension expression \rangle\}$
<code>\box_set_dp:cn</code>	Set the depth (below the baseline) of the $\langle box \rangle$ to the value of the $\{\langle dimension expression \rangle\}$. This is a global assignment.
Updated: 2011-10-22	

<code>\box_set_ht:Nn</code>	<code>\box_set_ht:Nn</code> $\langle box \rangle$ $\{\langle dimension expression \rangle\}$
<code>\box_set_ht:cn</code>	Set the height (above the baseline) of the $\langle box \rangle$ to the value of the $\{\langle dimension expression \rangle\}$. This is a global assignment.
Updated: 2011-10-22	

<code>\box_set_wd:Nn</code>	<code>\box_set_wd:Nn</code> $\langle box \rangle$ $\{\langle dimension expression \rangle\}$
<code>\box_set_wd:cn</code>	Set the width of the $\langle box \rangle$ to the value of the $\{\langle dimension expression \rangle\}$. This is a global assignment.
Updated: 2011-10-22	

4 Box conditionals

<code>\box_if_empty_p:N</code> *	<code>\box_if_empty_p:N</code> $\langle box \rangle$
<code>\box_if_empty_p:c</code> *	<code>\box_if_empty:N</code> $\langle box \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\box_if_empty:NTF</code> *	Tests if $\langle box \rangle$ is a empty (equal to <code>\c_empty_box</code>).
<code>\box_if_empty:cTF</code> *	

<code>\box_if_horizontal_p:N</code> *	<code>\box_if_horizontal_p:N</code> $\langle box \rangle$
<code>\box_if_horizontal_p:c</code> *	<code>\box_if_horizontal:N</code> $\langle box \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\box_if_horizontal:NTF</code> *	Tests if $\langle box \rangle$ is a horizontal box.
<code>\box_if_horizontal:cTF</code> *	

<code>\box_if_vertical_p:N</code>	★	<code>\box_if_vertical_p:N</code>	$\langle box \rangle$
<code>\box_if_vertical_p:c</code>	★	<code>\box_if_vertical:NTF</code>	$\langle box \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\box_if_vertical:NTF</code>	★		Tests if $\langle box \rangle$ is a vertical box.
<code>\box_if_vertical:cTF</code>	★		

5 The last box inserted

<code>\box_set_to_last:N</code>	<code>\box_set_to_last:N</code>	$\langle box \rangle$
<code>\box_set_to_last:c</code>		Sets the $\langle box \rangle$ equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the $\langle box \rangle$ is always void as it is not possible to recover the last added item.
<code>\box_gset_to_last:N</code>		
<code>\box_gset_to_last:c</code>		

6 Constant boxes

<code>\c_empty_box</code>	This is a permanently empty box, which is neither set as horizontal nor vertical.
Updated: 2012-11-04	TeXhackers note: At the TeX level this is a void box.

7 Scratch boxes

<code>\l_tmpa_box</code>	Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_box</code>	
Updated: 2012-11-04	

<code>\g_tmpa_box</code>	Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_box</code>	

8 Viewing box contents

<code>\box_show:N</code>	<code>\box_show:N</code>	$\langle box \rangle$
<code>\box_show:c</code>		Shows full details of the content of the $\langle box \rangle$ in the terminal.
Updated: 2012-05-11		
<code>\box_show:Nnn</code>	<code>\box_show:Nnn</code>	$\langle box \rangle$ $\{\langle intexpr_1 \rangle\}$ $\{\langle intexpr_2 \rangle\}$
<code>\box_show:cnn</code>		Display the contents of $\langle box \rangle$ in the terminal, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.
New: 2012-05-11		

<code>\box_log:N</code>	<code>\box_log:N <box></code>
<code>\box_log:c</code>	Writes full details of the content of the $\langle box \rangle$ to the log.
New: 2012-05-11	

<code>\box_log:Nnn</code>	<code>\box_log:Nnn <box> {\intexpr_1} {\intexpr_2}</code>
<code>\box_log:cnn</code>	Writes the contents of $\langle box \rangle$ to the log, showing the first $\langle intexpr_1 \rangle$ items of the box, and descending into $\langle intexpr_2 \rangle$ group levels.
New: 2012-05-11	

9 Boxes and color

All L^AT_EX3 boxes are “color safe”: a color set inside the box stops applying after the end of the box has occurred.

10 Horizontal mode boxes

<code>\hbox:n</code>	<code>\hbox:n {\contents}</code>
Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a horizontal box of natural width and then includes this box in the current list for typesetting.

<code>\hbox_to_wd:nn</code>	<code>\hbox_to_wd:nn {\dimexpr} {\contents}</code>
Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a horizontal box of width $\langle dimexpr \rangle$ and then includes this box in the current list for typesetting.

<code>\hbox_to_zero:n</code>	<code>\hbox_to_zero:n {\contents}</code>
Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a horizontal box of zero width and then includes this box in the current list for typesetting.

<code>\hbox_set:Nn</code>	<code>\hbox_set:Nn <box> {\contents}</code>
<code>\hbox_set:cn</code>	Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$.
<code>\hbox_gset:Nn</code>	
<code>\hbox_gset:cn</code>	
Updated: 2017-04-05	

<code>\hbox_set_to_wd:Nnn</code>	<code>\hbox_set_to_wd:Nnn $\langle box \rangle$ $\{\langle dimexpr \rangle\}$ $\{\langle contents \rangle\}$</code>
<code>\hbox_set_to_wd:cnn</code>	Typesets the $\langle contents \rangle$ to the width given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$.
<code>\hbox_gset_to_wd:Nnn</code>	
<code>\hbox_gset_to_wd:cnn</code>	
<hr/>	
Updated: 2017-04-05	

<code>\hbox_overlap_right:n</code>	<code>\hbox_overlap_right:n {\contents}</code>
Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material protrudes to the right of the insertion point.

<hr/> <code>\hbox_overlap_left:n</code> <hr/>	<code>\hbox_overlap_left:n {\langle contents \rangle}</code>
Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material protrudes to the left of the insertion point.
<hr/> <code>\hbox_set:Nw</code> <code>\hbox_set:cw</code> <code>\hbox_set_end:</code> <code>\hbox_gset:Nw</code> <code>\hbox_gset:cw</code> <code>\hbox_gset_end:</code> <hr/>	<code>\hbox_set:Nw \langle box \rangle \langle contents \rangle \hbox_set_end:</code> Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$. In contrast to <code>\hbox_set:Nn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.
Updated: 2017-04-05	
<hr/> <code>\hbox_set_to_wd:Nnw</code> <code>\hbox_set_to_wd:cnw</code> <code>\hbox_gset_to_wd:Nnw</code> <code>\hbox_gset_to_wd:cnw</code> <hr/>	<code>\hbox_set_to_wd:Nnw \langle box \rangle {\langle dimexpr \rangle} \langle contents \rangle \hbox_set_end:</code> Typesets the $\langle contents \rangle$ to the width given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$. In contrast to <code>\hbox_set_to_wd:Nnn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument
New: 2017-06-08	
<hr/> <code>\hbox_unpack:N</code> <code>\hbox_unpack:c</code> <hr/>	<code>\hbox_unpack:N \langle box \rangle</code> Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.
	TeXhackers note: This is the TeX primitive <code>\unhcopy</code> .
<hr/> <code>\hbox_unpack_clear:N</code> <code>\hbox_unpack_clear:c</code> <hr/>	<code>\hbox_unpack_clear:N \langle box \rangle</code> Unpacks the content of the horizontal $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The $\langle box \rangle$ is then cleared globally.
	TeXhackers note: This is the TeX primitive <code>\unhbox</code> .

11 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are `_top` boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

<hr/> <code>\vbox:n</code> <hr/>	<code>\vbox:n {\langle contents \rangle}</code>
Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a vertical box of natural height and includes this box in the current list for typesetting.
<hr/> <code>\vbox_top:n</code> <hr/>	<code>\vbox_top:n {\langle contents \rangle}</code>
Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the <i>first</i> item added to the box.

<code>\vbox_to_ht:nn</code>	<code>\vbox_to_ht:nn {<dimexpr>} {<contents>}</code>
-----------------------------	--

Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a vertical box of height $\langle dimexpr \rangle$ and then includes this box in the current list for typesetting.
---------------------	---

<code>\vbox_to_zero:n</code>	<code>\vbox_to_zero:n {<contents>}</code>
------------------------------	---

Updated: 2017-04-05	Typesets the $\langle contents \rangle$ into a vertical box of zero height and then includes this box in the current list for typesetting.
---------------------	--

<code>\vbox_set:Nn</code>	<code>\vbox_set:Nn <box> {<contents>}</code>
---------------------------	--

<code>\vbox_set:cn</code> <code>\vbox_gset:Nn</code> <code>\vbox_gset:cn</code>	Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$.
---	---

Updated: 2017-04-05	
---------------------	--

<code>\vbox_set_top:Nn</code>	<code>\vbox_set_top:Nn <box> {<contents>}</code>
-------------------------------	--

<code>\vbox_set_top:cn</code> <code>\vbox_gset_top:Nn</code> <code>\vbox_gset_top:cn</code>	Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. The baseline of the box is equal to that of the <i>first</i> item added to the box.
---	---

Updated: 2017-04-05	
---------------------	--

<code>\vbox_set_to_ht:Nnn</code>	<code>\vbox_set_to_ht:Nnn <box> {<dimexpr>} {<contents>}</code>
----------------------------------	---

<code>\vbox_set_to_ht:cn</code> <code>\vbox_gset_to_ht:Nnn</code> <code>\vbox_gset_to_ht:cn</code>	Typesets the $\langle contents \rangle$ to the height given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$.
--	--

Updated: 2017-04-05	
---------------------	--

<code>\vbox_set:Nw</code>	<code>\vbox_set:Nw <box> <contents> \vbox_set_end:</code>
---------------------------	---

<code>\vbox_set:cw</code> <code>\vbox_set_end:</code> <code>\vbox_gset:Nw</code> <code>\vbox_gset:cw</code> <code>\vbox_gset_end:</code>	Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. In contrast to <code>\vbox_set:Nn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.
--	--

Updated: 2017-04-05	
---------------------	--

<code>\vbox_set_to_ht:Nnw</code>	<code>\vbox_set_to_wd:Nnw <box> {<dimexpr>} <contents> \vbox_set_end:</code>
----------------------------------	--

<code>\vbox_set_to_ht:cnw</code> <code>\vbox_gset_to_ht:Nnw</code> <code>\vbox_gset_to_ht:cnw</code>	Typesets the $\langle contents \rangle$ to the height given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$. In contrast to <code>\vbox_set_to_ht:Nnn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument
--	---

New: 2017-06-08	
-----------------	--

<code>\vbox_set_split_to_ht:NNn</code>	<code>\vbox_set_split_to_ht:NNn <box₁₂</code>
--	---

Updated: 2011-10-22	Sets $\langle box_1 \rangle$ to contain material to the height given by the $\langle dimexpr \rangle$ by removing content from the top of $\langle box_2 \rangle$ (which must be a vertical box).
---------------------	---

T_EXhackers note: This is the T_EX primitive `\vsplit`.

<hr/> <code>\vbox_unpack:N</code>	<code>\vbox_unpack:N <box></code>
<hr/> <code>\vbox_unpack:c</code>	Unpacks the content of the vertical <code><box></code> , retaining any stretching or shrinking applied when the <code><box></code> was set.

T_EXhackers note: This is the T_EX primitive `\unvcopy`.

<hr/> <code>\vbox_unpack_clear:N</code>	<code>\vbox_unpack:N <box></code>
<hr/> <code>\vbox_unpack_clear:c</code>	Unpacks the content of the vertical <code><box></code> , retaining any stretching or shrinking applied when the <code><box></code> was set. The <code><box></code> is then cleared globally.

T_EXhackers note: This is the T_EX primitive `\unvbox`.

12 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in T_EX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

<hr/> <code>\box_autosize_to_wd_and_ht:Nnn</code>	<code>\box_autosize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}</code>
<hr/> <code>\box_autosize_to_wd_and_ht:cnn</code>	

New: 2017-04-04

Resizes the `<box>` to fit within the given `<x-size>` (horizontally) and `<y-size>` (vertically); both of the sizes are dimension expressions. The `<y-size>` is the height only: it does not include any depth. The updated `<box>` is an `hbox`, irrespective of the nature of the `<box>` before the resizing is applied. The final size of the `<box>` is the smaller of `{<x-size>}` and `{<y-size>}`, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the `<box>` to be reversed in direction, but the reference point of the `<box>` is unchanged. Thus a negative `<y-size>` results in the `<box>` having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current T_EX group level.

<hr/> <code>\box_autosize_to_wd_and_ht_plus_dp:Nnn</code>	<code>\box_autosize_to_wd_and_ht_plus_dp:Nnn <box> {<x-size>}</code>
<hr/> <code>\box_autosize_to_wd_and_ht_plus_dp:cnn</code>	<code>{<y-size>}</code>

New: 2017-04-04

Resizes the `<box>` to fit within the given `<x-size>` (horizontally) and `<y-size>` (vertically); both of the sizes are dimension expressions. The `<y-size>` is the total vertical size (height plus depth). The updated `<box>` is an `hbox`, irrespective of the nature of the `<box>` before the resizing is applied. The final size of the `<box>` is the smaller of `{<x-size>}` and `{<y-size>}`, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the `<box>` to be reversed in direction, but the reference point of the `<box>` is unchanged. Thus a negative `<y-size>` results in the `<box>` having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current T_EX group level.

`\box_resize_to_ht:Nn` `\box_resize_to_ht:Nn <box> {<y-size>}`

`\box_resize_to_ht:cn`

Resizes the $\langle box \rangle$ to $\langle y-size \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y-size \rangle$ is a dimension expression. The $\langle y-size \rangle$ is the height only: it does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current \TeX group level.

`\box_resize_to_ht_plus_dp:Nn` `\box_resize_to_ht_plus_dp:Nn <box> {<y-size>}`

`\box_resize_to_ht_plus_dp:cn`

Resizes the $\langle box \rangle$ to $\langle y-size \rangle$ (vertically), scaling the horizontal size by the same amount; $\langle y-size \rangle$ is a dimension expression. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle y-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current \TeX group level.

`\box_resize_to_wd:Nn` `\box_resize_to_wd:Nn <box> {<x-size>}`

`\box_resize_to_wd:cn`

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally), scaling the vertical size by the same amount; $\langle x-size \rangle$ is a dimension expression. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative $\langle x-size \rangle$ causes the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle x-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current \TeX group level.

`\box_resize_to_wd_and_ht:Nnn` `\box_resize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}`

`\box_resize_to_wd_and_ht:cnn`

New: 2014-07-03

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the height only and does not include any depth. The updated $\langle box \rangle$ is an `hbox`, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current \TeX group level.

<code>\box_resize_to_wd_and_ht_plus_dp:Nnn</code>	<code>\box_resize_to_wd_and_ht_plus_dp:Nnn <box> {(x-size)} {(y-size)}</code>
<code>\box_resize_to_wd_and_ht_plus_dp:cnn</code>	

New: 2017-04-06

Resizes the $\langle box \rangle$ to $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically): both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the total vertical size (height plus depth). The updated $\langle box \rangle$ is an **hbox**, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-size \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current \TeX group level.

<code>\box_rotate:Nn</code>	<code>\box_rotate:Nn <box> {(angle)}</code>
<code>\box_rotate:cn</code>	

Rotates the $\langle box \rangle$ by $\langle angle \rangle$ (in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated $\langle box \rangle$ is an **hbox**, irrespective of the nature of the $\langle box \rangle$ before the rotation is applied. The rotation applies within the current \TeX group level.

<code>\box_scale:Nnn</code>	<code>\box_scale:Nnn <box> {(x-scale)} {(y-scale)}</code>
<code>\box_scale:cnn</code>	

Scales the $\langle box \rangle$ by factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated $\langle box \rangle$ is an **hbox**, irrespective of the nature of the $\langle box \rangle$ before the scaling is applied. Negative scalings cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ is unchanged. Thus a negative $\langle y-scale \rangle$ results in the $\langle box \rangle$ having a depth dependent on the height of the original and *vice versa*. The resizing applies within the current \TeX group level.

13 Primitive box conditionals

<code>\if_hbox:N</code> ★	<code>\if_hbox:N <box></code> <code> <true code></code> <code>\else:</code> <code> <false code></code> <code>\fi:</code>
---------------------------	--

Tests if $\langle box \rangle$ is a horizontal box.

\TeX hackers note: This is the \TeX primitive `\ifhbox`.

<code>\if_vbox:N</code> ★	<code>\if_vbox:N <box></code> <code> <true code></code> <code>\else:</code> <code> <false code></code> <code>\fi:</code>
---------------------------	--

Tests if $\langle box \rangle$ is a vertical box.

\TeX hackers note: This is the \TeX primitive `\ifvbox`.

`\if_box_empty:N` ★ `\if_box_empty:N` $\langle box \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle false\ code \rangle$
`\fi:`
Tests if $\langle box \rangle$ is an empty (void) box.

TeXhackers note: This is the TeX primitive `\ifvoid`.

Part XXVII

The l3coffins package

Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the xcoffins module (in the l3experimental bundle).

1 Creating and initialising coffins

<code>\coffin_new:N</code>
<code>\coffin_new:c</code>
New: 2011-08-17

`\coffin_new:N` $\langle coffin \rangle$

Creates a new $\langle coffin \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle coffin \rangle$ is initially empty.

<code>\coffin_clear:N</code>
<code>\coffin_clear:c</code>
New: 2011-08-17

`\coffin_clear:N` $\langle coffin \rangle$

Clears the content of the $\langle coffin \rangle$ within the current T_EX group level.

<code>\coffin_set_eq:NN</code>
<code>\coffin_set_eq:(Nc cN cc)</code>
New: 2011-08-17

`\coffin_set_eq:NN` $\langle coffin_1 \rangle$ $\langle coffin_2 \rangle$

Sets both the content and poles of $\langle coffin_1 \rangle$ equal to those of $\langle coffin_2 \rangle$ within the current T_EX group level.

<code>\coffin_if_exist_p:N</code> ★
<code>\coffin_if_exist_p:c</code> ★
<code>\coffin_if_exist:NTF</code> ★
<code>\coffin_if_exist:cTF</code> ★
New: 2012-06-20

`\coffin_if_exist_p:N` $\langle box \rangle$

`\coffin_if_exist:NTF` $\langle box \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests whether the $\langle coffin \rangle$ is currently defined.

2 Setting coffin content and poles

All coffin functions create and manipulate coffins locally within the current T_EX group level.

<code>\hcoffin_set:Nn</code>
<code>\hcoffin_set:cn</code>
New: 2011-08-17
Updated: 2011-09-03

`\hcoffin_set:Nn` $\langle coffin \rangle$ $\{\langle material \rangle\}$

Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.

<code>\hcoffin_set:Nw</code>
<code>\hcoffin_set:cw</code>
<code>\hcoffin_set_end:</code>
New: 2011-09-10

`\hcoffin_set:Nw` $\langle coffin \rangle$ $\langle material \rangle$ `\hcoffin_set_end:`

Typesets the $\langle material \rangle$ in horizontal mode, storing the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

`\vcoffin_set:Nnn`
`\vcoffin_set:cnn`
 New: 2011-08-17
 Updated: 2012-05-22

`\vcoffin_set:Nnn` $\langle coffin \rangle$ $\{\langle width \rangle\}$ $\{\langle material \rangle\}$
 Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material.

`\vcoffin_set:Nnw`
`\vcoffin_set:cnw`
`\vcoffin_set_end:`
 New: 2011-09-10
 Updated: 2012-05-22

`\vcoffin_set:Nnw` $\langle coffin \rangle$ $\{\langle width \rangle\}$ $\langle material \rangle$ `\vcoffin_set_end:`
 Typesets the $\langle material \rangle$ in vertical mode constrained to the given $\langle width \rangle$ and stores the result in the $\langle coffin \rangle$. The standard poles for the $\langle coffin \rangle$ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

`\coffin_set_horizontal_pole:Nnn` `\coffin_set_horizontal_pole:Nnn` $\langle coffin \rangle$
`\coffin_set_horizontal_pole:cnn` $\{\langle pole \rangle\}$ $\{\langle offset \rangle\}$
 New: 2012-07-20

Sets the $\langle pole \rangle$ to run horizontally through the $\langle coffin \rangle$. The $\langle pole \rangle$ is placed at the $\langle offset \rangle$ from the bottom edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

`\coffin_set_vertical_pole:Nnn` `\coffin_set_vertical_pole:Nnn` $\langle coffin \rangle$ $\{\langle pole \rangle\}$ $\{\langle offset \rangle\}$
`\coffin_set_vertical_pole:cnn`
 New: 2012-07-20

Sets the $\langle pole \rangle$ to run vertically through the $\langle coffin \rangle$. The $\langle pole \rangle$ is placed at the $\langle offset \rangle$ from the left-hand edge of the bounding box of the $\langle coffin \rangle$. The $\langle offset \rangle$ should be given as a dimension expression.

3 Joining and using coffins

`\coffin_attach:NnnNnnnn` `\coffin_attach:NnnNnnnn`
`\coffin_attach:(cnnNnnnn|Nnnncnnnn|cnnncnnnn)` $\langle coffin_1 \rangle$ $\{\langle coffin_1-pole_1 \rangle\}$ $\{\langle coffin_1-pole_2 \rangle\}$
 $\langle coffin_2 \rangle$ $\{\langle coffin_2-pole_1 \rangle\}$ $\{\langle coffin_2-pole_2 \rangle\}$
 $\{\langle x-offset \rangle\}$ $\{\langle y-offset \rangle\}$

This function attaches $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ is not altered, *i.e.* $\langle coffin_2 \rangle$ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

```
\coffin_join:NnnNnnnn
\coffin_join:(cnnNnnnn|Nnnncnnnn|cnnncnnnn)
```

```
\coffin_join:NnnNnnnn
  <coffin_1> {<coffin_1-pole_1>} {<coffin_1-pole_2>}
  <coffin_2> {<coffin_2-pole_1>} {<coffin_2-pole_2>}
  {<x-offset>} {<y-offset>}
```

This function joins $\langle coffin_2 \rangle$ to $\langle coffin_1 \rangle$ such that the bounding box of $\langle coffin_1 \rangle$ may expand. The new bounding box covers the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating $\langle handle_1 \rangle$, the point of intersection of $\langle coffin_1-pole_1 \rangle$ and $\langle coffin_1-pole_2 \rangle$, and $\langle handle_2 \rangle$, the point of intersection of $\langle coffin_2-pole_1 \rangle$ and $\langle coffin_2-pole_2 \rangle$. $\langle coffin_2 \rangle$ is then attached to $\langle coffin_1 \rangle$ such that the relationship between $\langle handle_1 \rangle$ and $\langle handle_2 \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions.

```
\coffin_typeset:Nnnnn
\coffin_typeset:cnnnn
```

Updated: 2012-07-20

```
\coffin_typeset:Nnnnn <coffin> {<pole_1>} {<pole_2>}
  {<x-offset>} {<y-offset>}
```

Typesetting is carried out by first calculating $\langle handle \rangle$, the point of intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the $\langle handle \rangle$ is described by the $\langle x-offset \rangle$ and $\langle y-offset \rangle$. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

4 Measuring coffins

```
\coffin_dp:N
\coffin_dp:c
```

```
\coffin_dp:N <coffin>
```

Calculates the depth (below the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

```
\coffin_ht:N
\coffin_ht:c
```

```
\coffin_ht:N <coffin>
```

Calculates the height (above the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

```
\coffin_wd:N
\coffin_wd:c
```

```
\coffin_wd:N <coffin>
```

Calculates the width of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

5 Coffin diagnostics

```
\coffin_display_handles:Nn
\coffin_display_handles:cn
```

Updated: 2011-09-02

```
\coffin_display_handles:Nn <coffin> {<color>}
```

This function first calculates the intersections between all of the $\langle poles \rangle$ of the $\langle coffin \rangle$ to give a set of $\langle handles \rangle$. It then prints the $\langle coffin \rangle$ at the current location in the source, with the position of the $\langle handles \rangle$ marked on the coffin. The $\langle handles \rangle$ are labelled as part of this process: the locations of the $\langle handles \rangle$ and the labels are both printed in the $\langle color \rangle$ specified.

<hr/> <code>\coffin_mark_handle:Nnnn</code> <code>\coffin_mark_handle:cnnn</code> <hr/> Updated: 2011-09-02 <hr/>	<code>\coffin_mark_handle:Nnnn <coffin> {<pole₁>} {<pole₂>} {<color>}</code> <p>This function first calculates the $\langle handle \rangle$ for the $\langle coffin \rangle$ as defined by the intersection of $\langle pole_1 \rangle$ and $\langle pole_2 \rangle$. It then marks the position of the $\langle handle \rangle$ on the $\langle coffin \rangle$. The $\langle handle \rangle$ are labelled as part of this process: the location of the $\langle handle \rangle$ and the label are both printed in the $\langle color \rangle$ specified.</p>
<hr/> <code>\coffin_show_structure:N</code> <code>\coffin_show_structure:c</code> <hr/> Updated: 2015-08-01 <hr/>	<code>\coffin_show_structure:N <coffin></code> <p>This function shows the structural information about the $\langle coffin \rangle$ in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.</p> <p>Notice that the poles of a coffin are defined by four values: the x and y co-ordinates of a point that the pole passes through and the x- and y-components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.</p>
<hr/> <code>\coffin_log_structure:N</code> <code>\coffin_log_structure:c</code> <hr/> New: 2014-08-22 Updated: 2015-08-01 <hr/>	<code>\coffin_log_structure:N <coffin></code> <p>This function writes the structural information about the $\langle coffin \rangle$ in the log file. See also <code>\coffin_show_structure:N</code> which displays the result in the terminal.</p>

6 Constants and variables

<hr/> <code>\c_empty_coffin</code> <hr/>	A permanently empty coffin.
<hr/> <code>\l_tmpa_coffin</code> <code>\l_tmpp_coffin</code> <hr/> New: 2012-06-19 <hr/>	Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Part XXVIII

The l3color-base package

Color support

This module provides support for color in L^AT_EX3. At present, the material here is mainly intended to support a small number of low-level requirements in other l3kernel modules.

1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

```
\color_group_begin:  
\color_group_end:
```

New: 2011-09-03

```
\color_group_begin:  
...  
\color_group_end:
```

Creates a color group: one used to “trap” color settings.

```
\color_ensure_current:
```

New: 2011-09-03

```
\color_ensure_current:
```

Ensures that material inside a box uses the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin: ... \color_group_end:` group.

Part XXIX

The l3luatex package: LuaTeX-specific functions

The LuaTeX engine provides access to the Lua programming language, and with it access to the “internals” of TeX. In order to use this within the framework provided here, a family of functions is available. When used with pdfTeX, pTeX, upTeX or XeTeX these raise an error: use `\sys_if_engine luatex:T` to avoid this. Details on using Lua with the LuaTeX engine are given in the LuaTeX manual.

1 Breaking out to Lua

<code>\lua_now_x:n</code>	★
<code>\lua_now:n</code>	★

New: 2015-06-29

`\lua_now:n` $\{ \langle token\ list \rangle \}$

The $\langle token\ list \rangle$ is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting $\langle Lua\ input \rangle$ is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter executes the $\langle Lua\ input \rangle$ immediately, and in an expandable manner.

In the case of the `\lua_now_x:n` version the input is fully expanded by TeX in an x-type manner *but* the function remains fully expandable.

TeXhackers note: `\lua_now_x:n` is a macro wrapper around `\directlua:` when LuaTeX is in use two expansions are required to yield the result of the Lua code.

<code>\lua_shipout_x:n</code>
<code>\lua_shipout:n</code>

New: 2015-06-30

`\lua_shipout:n` $\{ \langle token\ list \rangle \}$

The $\langle token\ list \rangle$ is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting $\langle Lua\ input \rangle$ is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the $\langle Lua\ input \rangle$ during the page-building routine: no TeX expansion of the $\langle Lua\ input \rangle$ will occur at this stage.

In the case of the `\lua_shipout_x:n` version the input is fully expanded by TeX in an x-type manner during the shipout operation.

TeXhackers note: At a TeX level, the $\langle Lua\ input \rangle$ is stored as a “whatsit”.

<code>\lua_escape_x:n</code>	★	<code>\lua_escape:n {⟨token list⟩}</code>
------------------------------	---	---

<code>\lua_escape:n</code>	★
----------------------------	---

New: 2015-06-29

Converts the *⟨token list⟩* such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to `\n` and `\r`, respectively.

In the case of the `\lua_escape_x:n` version the input is fully expanded by $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ in an *x*-type manner *but* the function remains fully expandable.

$\mathrm{T}_{\mathrm{E}}\mathrm{X}$ hackers note: `\lua_escape_x:n` is a macro wrapper around `\luaescapestring`: when $\mathrm{LuaT}_{\mathrm{E}}\mathrm{X}$ is in use two expansions are required to yield the result of the Lua code.

2 Lua interfaces

As well as interfaces for $\mathrm{T}_{\mathrm{E}}\mathrm{X}$, there are a small number of Lua functions provided here.

<code>l3kernel</code>

All public interfaces provided by the module are stored within the `l3kernel` table.

<code>l3kernel.charcat</code>

`l3kernel.charcat(⟨charcode⟩, ⟨catcode⟩)`

Constructs a character of *⟨charcode⟩* and *⟨catcode⟩* and returns the result to $\mathrm{T}_{\mathrm{E}}\mathrm{X}$.

<code>l3kernel.filemdfivesum</code>

`l3kernel.filemdfivesum(⟨file⟩)`

Returns the of the MD5 sum of the file contents read as bytes; note that the result will depend on the nature of the line endings used in the file, in contrast to normal $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ behaviour. If the *⟨file⟩* is not found, nothing is returned with *no error raised*.

<code>l3kernel.filemoddate</code>

`l3kernel.filemoddate(⟨file⟩)`

Returns the of the date/time of last modification of the *⟨file⟩* in the format

`D:⟨year⟩⟨month⟩⟨day⟩⟨hour⟩⟨minute⟩⟨second⟩⟨offset⟩`

where the latter may be `Z` (UTC) or *⟨plus-minus⟩⟨hours⟩'⟨minutes⟩'*. If the *⟨file⟩* is not found, nothing is returned with *no error raised*.

<code>l3kernel.filesize</code>

`l3kernel.filesize(⟨file⟩)`

Returns the size of the *⟨file⟩* in bytes. If the *⟨file⟩* is not found, nothing is returned with *no error raised*.

<code>l3kernel.strcmp</code>

`l3kernel.strcmp(⟨str one⟩, ⟨str two⟩)`

Compares the two strings and returns 0 to $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ if the two are identical.

Part XXX

The l3unicode package: Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. At present, it provides no public functions.

Part XXXI

The l3candidates package

Experimental additions to l3kernel

1 Important notice

This module provides a space in which functions can be added to l3kernel (expl3) while still being experimental.

As such, the functions here may not remain in their current form, or indeed at all, in l3kernel in the future.

In contrast to the material in l3experimental, the functions here are all *small* additions to the kernel. We encourage programmers to test them out and report back on the **LaTeX-L** mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.
- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the **LaTeX-L** mailing list. This way it becomes easier to coordinate any updates necessary without issues for the users of your package.
- Discussing and understanding use cases for a particular addition or concept also helps to ensure that we provide the right interfaces in the final version so please give us feedback if you consider a certain candidate function useful (or not).

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are **not necessarily stable** and we may have to adjust them!

2 Additions to l3basics

`\debug_on:n`
`\debug_off:n`

New: 2017-07-16
Updated: 2017-08-02

`\debug_on:n { <comma-separated list> }`
`\debug_off:n { <comma-separated list> }`

Turn on and off within a group various debugging code, some of which is also available as expl3 load-time options. The items that can be used in the *<list>* are

- **check-declarations** that checks all expl3 variables used were previously declared and that local/global variables (based on their name or on their first assignment) are only locally/globally assigned;
- **check-expressions** that checks integer, dimension, skip, and muskip expressions are not terminated prematurely;
- **deprecation** that makes soon-to-be-deprecated commands produce errors;
- **log-functions** that logs function definitions;

Providing these as switches rather than options allows testing code even if it relies on other packages: load all other packages, call `\debug_on:n`, and load the code that one is interested in testing. These functions can only be used in L^AT_EX 2_ε package mode loaded with **enable-debug** or another option implying it.

`\debug_suspend:`
`\debug_resume:`

New: 2017-11-28

`\debug_suspend: ... \debug_resume:`

Suppress (locally) errors and logging from **debug** commands, except for the **deprecation** errors or warnings. These pairs of commands can be nested. This can be used around pieces of code that are known to fail checks, if such failures should be ignored. See for instance l3coffins.

`\mode_leave_vertical:`

New: 2017-07-04

`\mode_leave_vertical:`

Ensures that T_EX is not in vertical (inter-paragraph) mode. In horizontal or math mode this command has no effect, in vertical mode it switches to horizontal mode, and inserts a box of width `\parindent`, followed by the `\everypar` token list.

T_EXhackers note: This results in the contents of the `\everypar` token register being inserted, after `\mode_leave_vertical:` is complete. Notice that in contrast to the L^AT_EX 2_ε `\leavevmode` approach, no box is used by the method implemented here.

`\use_x:n` ★

New: 2018-04-17

`\use_x:n {<token list>}`

Fully expands the *<token list>* in an x-type manner, *but* the function remains fully expandable, and parameter character (usually #) need not be doubled.

T_EXhackers note: `\use_x:n` the a wrapper around the primitive `\expanded:` it requires two expansions to complete its action.

3 Additions to l3box

3.1 Viewing part of a box

<code>\box_clip:N</code>	<code>\box_clip:N <box></code>
<code>\box_clip:c</code>	

Clips the $\langle box \rangle$ in the output so that only material inside the bounding box is displayed in the output. The updated $\langle box \rangle$ is an hbox, irrespective of the nature of the $\langle box \rangle$ before the clipping is applied. The clipping applies within the current T_EX group level.

These functions require the L^AT_EX3 native drivers: they do not work with the L^AT_EX 2_ε graphics drivers!

T_EXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

<code>\box_trim:Nnnnn</code>	<code>\box_trim:Nnnnn <box> {<left>} {<bottom>} {<right>} {<top>}</code>
<code>\box_trim:cnnnn</code>	

Adjusts the bounding box of the $\langle box \rangle$ $\langle left \rangle$ is removed from the left-hand edge of the bounding box, $\langle right \rangle$ from the right-hand edge and so fourth. All adjustments are $\langle dimension expressions \rangle$. Material outside of the bounding box is still displayed in the output unless `\box_clip:N` is subsequently applied. The updated $\langle box \rangle$ is an hbox, irrespective of the nature of the $\langle box \rangle$ before the trim operation is applied. The adjustment applies within the current T_EX group level. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

<code>\box_viewport:Nnnnn</code>	<code>\box_viewport:Nnnnn <box> {<llx>} {<lly>} {<urx>} {<ury>}</code>
<code>\box_viewport:cnnnn</code>	

Adjusts the bounding box of the $\langle box \rangle$ such that it has lower-left co-ordinates ($\langle llx \rangle$, $\langle lly \rangle$) and upper-right co-ordinates ($\langle urx \rangle$, $\langle ury \rangle$). All four co-ordinate positions are $\langle dimension expressions \rangle$. Material outside of the bounding box is still displayed in the output unless `\box_clip:N` is subsequently applied. The updated $\langle box \rangle$ is an hbox, irrespective of the nature of the $\langle box \rangle$ before the viewport operation is applied. The adjustment applies within the current T_EX group level.

4 Additions to l3clist

<code>\clist_rand_item:N</code> ★	<code>\clist_rand_item:N <clist var></code>
<code>\clist_rand_item:c</code> ★	<code>\clist_rand_item:n {<comma list>}</code>
<code>\clist_rand_item:n</code> ★	

New: 2016-12-06

Selects a pseudo-random item of the $\langle comma list \rangle$. If the $\langle comma list \rangle$ has no item, the result is empty. This not yet available in X_YT_EX.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle item \rangle$ does not expand further when appearing in an x-type argument expansion.

5 Additions to l3coffins

<hr/> <code>\coffin_resize:Nnn</code> <code>\coffin_resize:cnn</code> <hr/>	<code>\coffin_resize:Nnn <coffin> {<width>} {<total-height>}</code> Resized the <code><coffin></code> to <code><width></code> and <code><total-height></code> , both of which should be given as dimension expressions.
<hr/> <code>\coffin_rotate:Nn</code> <code>\coffin_rotate:cn</code> <hr/>	<code>\coffin_rotate:Nn <coffin> {<angle>}</code> Rotates the <code><coffin></code> by the given <code><angle></code> (given in degrees counter-clockwise). This process rotates both the coffin content and poles. Multiple rotations do not result in the bounding box of the coffin growing unnecessarily.
<hr/> <code>\coffin_scale:Nnn</code> <code>\coffin_scale:cnn</code> <hr/>	<code>\coffin_scale:Nnn <coffin> {<x-scale>} {<y-scale>}</code> Scales the <code><coffin></code> by a factors <code><x-scale></code> and <code><y-scale></code> in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

6 Additions to l3expan

<hr/> <code>\prg_generate_conditional_variant:Nnn</code> <div>New: 2017-12-12</div> <hr/>	<code>\prg_generate_conditional_variant:Nnn \<name>:<arg spec></code> <code>{<variant argument specifiers>} {<condition specifiers>}</code> Defines argument-specifier variants of conditionals. This is equivalent to running <code>\cs_generate_variant:Nn <conditional> {<variant argument specifiers>}</code> on each <code><conditional></code> described by the <code><condition specifiers></code> . These base-form <code><conditionals></code> are obtained from the <code><name></code> and <code><arg spec></code> as described for <code>\prg_new_conditional:Npnn</code> , and they should be defined.
<hr/> <code>\exp_args_generate:n</code> <div>New: 2018-04-04</div> <hr/>	<code>\exp_args_generate:n {<variant argument specifiers>}</code> Defines <code>\exp_args:N<variant></code> functions for each <code><variant></code> given in the comma list <code>{<variant argument specifiers>}</code> . Each <code><variant></code> should consist of the letters N, c, n, V, v, o, f, x, p and the resulting function is protected if the letter x appears in the <code><variant></code> . This is only useful for cases where <code>\cs_generate_variant:Nn</code> is not applicable.

7 Additions to l3file

<hr/> <code>\file_get_md5five_hash:nN</code> <div>New: 2017-07-11</div> <hr/>	<code>\file_get_md5five_hash:nN {<file name>} <str var></code> Searches for <code><file name></code> using the current T _E X search path and the additional paths controlled by <code>\file_path_include:n</code> . If found, sets the <code><str var></code> to the MD5 sum generated from the content of the file. The file is read as bytes, which means that in contrast to most T _E X behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. Where the file is not found, the <code><str var></code> will be empty.
--	--

\file_get_size:nN

New: 2017-07-09

\file_get_size:nN $\{\langle file\ name\rangle\}$ $\langle str\ var\rangle$

Searches for $\langle file\ name\rangle$ using the current \TeX search path and the additional paths controlled by $\text{\file_path_include:n}$. If found, sets the $\langle str\ var\rangle$ to the size of the file in bytes. Where the file is not found, the $\langle str\ var\rangle$ will be empty.

\TeX hackers note: The $\text{X}_{\text{\TeX}}$ engine provides no way to implement this function.

\file_get_timestamp:nN

New: 2017-07-09

\file_get_timestamp:nN $\{\langle file\ name\rangle\}$ $\langle str\ var\rangle$

Searches for $\langle file\ name\rangle$ using the current \TeX search path and the additional paths controlled by $\text{\file_path_include:n}$. If found, sets the $\langle str\ var\rangle$ to the modification timestamp of the file in the form $D:\langle year\rangle\langle month\rangle\langle day\rangle\langle hour\rangle\langle minute\rangle\langle second\rangle\langle offset\rangle$, where the latter may be Z (UTC) or $\langle plus-minus\rangle\langle hours\rangle'\langle minutes\rangle'$. Where the file is not found, the $\langle str\ var\rangle$ will be empty.

\TeX hackers note: The $\text{X}_{\text{\TeX}}$ engine provides no way to implement this function.

\file_if_exist_input:n**\file_if_exist_input:nF**

New: 2014-07-02

\file_if_exist_input:n $\{\langle file\ name\rangle\}$ **\file_if_exist_input:nF** $\{\langle file\ name\rangle\}$ $\{\langle false\ code\rangle\}$

Searches for $\langle file\ name\rangle$ using the current \TeX search path and the additional paths controlled by $\text{\file_path_include:n}$. If found then reads in the file as additional \LaTeX source as described for \file_input:n , otherwise inserts the $\langle false\ code\rangle$. Note that these functions do not raise an error if the file is not found, in contrast to \file_input:n .

\file_input_stop:

New: 2017-07-07

\file_input_stop:

Ends the reading of a file started by \file_input:n or similar before the end of the file is reached. Where the file reading is being terminated due to an error, $\text{\msg_critical:nn(nn)}$ should be preferred.

\TeX hackers note: This function must be used on a line on its own: \TeX reads files line-by-line and so any additional tokens in the “current” line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn’t help as it is the line where it is used that counts!

8 Additions to **l3flag**

\flag_raise_if_clear:n ★

New: 2018-04-02

\flag_raise_if_clear:n $\{\langle flag\ name\rangle\}$

Ensures the $\langle flag\rangle$ is raised by making its height at least 1, locally.

9 Additions to **l3msg**

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using \msg_error:nnnnnn or other non-expandable commands from the previous section.

Despite having a similar interface as non-expandable messages, expandable errors must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. As a result, the message text and arguments are not expanded, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message is understandable even when truncated. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

<code>\msg_expandable_error:nnnnnn</code>	★	<code>\msg_expandable_error:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_expandable_error:nnffff</code>	★	
<code>\msg_expandable_error:nnnnn</code>	★	
<code>\msg_expandable_error:nnfff</code>	★	
<code>\msg_expandable_error:nnnn</code>	★	
<code>\msg_expandable_error:nnff</code>	★	
<code>\msg_expandable_error:nnn</code>	★	
<code>\msg_expandable_error:nnf</code>	★	
<code>\msg_expandable_error:nn</code>	★	

New: 2015-08-06

Issues an “Undefined error” message from T_EX itself using the undefined control sequence `\::error` then prints “! <module>: ”<error message>, which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.

<code>\msg_show_eval:Nn</code>	<code>\msg_show_eval:Nn <function> {<expression>}</code>
<code>\msg_log_eval:Nn</code>	

New: 2017-12-04

Shows or logs the <expression> (turned into a string), an equal sign, and the result of applying the <function> to the {<expression>} (with f-expansion). For instance, if the <function> is `\int_eval:n` and the <expression> is `1+2` then this logs `> 1+2=3`.

<code>\msg_show:nnnnnn</code>	<code>\msg_show:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>\msg_show:nnxxxx</code>	
<code>\msg_show:nnnnn</code>	
<code>\msg_show:nnxxx</code>	
<code>\msg_show:nnnn</code>	
<code>\msg_show:nnxx</code>	
<code>\msg_show:nnn</code>	
<code>\msg_show:nnx</code>	
<code>\msg_show:nn</code>	

New: 2017-12-04

Issues <module> information <message>, passing <arg one> to <arg four> to the text-creating functions. The information text is shown on the terminal and the T_EX run is interrupted in a manner similar to `\tl_show:n`. This is used in conjunction with `\msg_show_item:n` and similar functions to print complex variable contents completely. If the formatted text does not contain `>~` at the start of a line, an additional line `>~`. will be put at the end. In addition, a final period is added if not present.

<code>\msg_show_item:n</code>	★	<code>\seq_map_function:NN <seq> \msg_show_item:n</code>
<code>\msg_show_item_unbraced:n</code>	★	<code>\prop_map_function:NN <prop> \msg_show_item:nn</code>
<code>\msg_show_item:nn</code>	★	
<code>\msg_show_item_unbraced:nn</code>	★	

New: 2017-12-04

Used in the text of messages for `\msg_show:nnxxxx` to show or log a list of items or key–value pairs. The one-argument functions are used for sequences, clist or token lists and the others for property lists. These functions turn their arguments to strings.

10 Additions to l3prg

<code>\bool_const:Nn</code>	<code>\bool_const:Nn <boolean> {<boolexpr>}</code>
-----------------------------	--

<code>\bool_const:cn</code>

New: 2017-11-28

Creates a new constant $\langle boolean \rangle$ or raises an error if the name is already taken. The value of the $\langle boolean \rangle$ is set globally to the result of evaluating the $\langle boolexpr \rangle$.

11 Additions to l3prop

<code>\prop_count:N</code> ★	<code>\prop_count:N <property list></code>
------------------------------	--

<code>\prop_count:c</code> ★

Leaves the number of key–value pairs in the $\langle property list \rangle$ in the input stream as an $\langle integer denotation \rangle$.

<code>\prop_map_tokens:Nn</code> ★

<code>\prop_map_tokens:cn</code> ★

`\prop_map_tokens:Nn <property list> {<code>}`

Analogue of `\prop_map_function:NN` which maps several tokens instead of a single function. The $\langle code \rangle$ receives each key–value pair in the $\langle property list \rangle$ as two trailing brace groups. For instance,

`\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }`

expands to the value corresponding to `mykey`: for each pair in `\l_my_prop` the function `\str_if_eq:nnT` receives `mykey`, the $\langle key \rangle$ and the $\langle value \rangle$ as its three arguments. For that specific task, `\prop_item:Nn` is faster.

<code>\prop_rand_key_value:N</code> ★

<code>\prop_rand_key_value:c</code> ★

New: 2016-12-06

`\prop_rand_key_value:N <prop var>`

Selects a pseudo-random key–value pair from the $\langle property list \rangle$ and returns $\{\langle key \rangle\}$ and $\{\langle value \rangle\}$. If the $\langle property list \rangle$ is empty the result is empty. This is not yet available in \TeX .

\TeX hackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the $\langle value \rangle$ does not expand further when appearing in an `x`-type argument expansion.

<code>\prop_set_from_keyval:Nn</code>

<code>\prop_set_from_keyval:cn</code>

<code>\prop_gset_from_keyval:Nn</code>
--

<code>\prop_gset_from_keyval:cn</code>
--

New: 2017-11-28

`\prop_set_from_keyval:Nn <prop var>`

```
{
  <key1> = <value1> ,
  <key2> = <value2> , ...
}
```

Sets $\langle prop var \rangle$ to contain key–value pairs given in the second argument.

<code>\prop_const_from_keyval:Nn</code>

<code>\prop_const_from_keyval:cn</code>

New: 2017-11-28

`\prop_const_from_keyval:Nn <prop var>`

```
{
  <key1> = <value1> ,
  <key2> = <value2> , ...
}
```

Creates a new constant $\langle prop var \rangle$ or raises an error if the name is already taken. The $\langle prop var \rangle$ is set globally to contain key–value pairs given in the second argument.

12 Additions to l3seq

<code>\seq_mapthread_function:NNN</code>	☆	<code>\seq_mapthread_function:NNN <seq₁> <seq₂> <function></code>
<code>\seq_mapthread_function:(NcN cNN ccN)</code>	☆	

Applies *<function>* to every pair of items *<seq₁-item>*–*<seq₂-item>* from the two sequences, returning items from both sequences from left to right. The *<function>* receives two *n*-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

<code>\seq_set_filter:NNn</code>	<code>\seq_set_filter:NNn <sequence₁> <sequence₂> {<inline boolexpr>}</code>
<code>\seq_gset_filter:NNn</code>	

Evaluates the *<inline boolexpr>* for every *<item>* stored within the *<sequence₂>*. The *<inline boolexpr>* receives the *<item>* as #1. The sequence of all *<items>* for which the *<inline boolexpr>* evaluated to `true` is assigned to *<sequence₁>*.

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

<code>\seq_set_map:NNn</code>	<code>\seq_set_map:NNn <sequence₁> <sequence₂> {<inline function>}</code>
<code>\seq_gset_map:NNn</code>	

New: 2011-12-22

Applies *<inline function>* to every *<item>* stored within the *<sequence₂>*. The *<inline function>* should consist of code which will receive the *<item>* as #1. The sequence resulting from *x*-expanding *<inline function>* applied to each *<item>* is assigned to *<sequence₁>*. As such, the code in *<inline function>* should be expandable.

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

<code>\seq_rand_item:N</code>	★	<code>\seq_rand_item:N <seq var></code>
<code>\seq_rand_item:c</code>	★	

New: 2016-12-06

Selects a pseudo-random item of the *<sequence>*. If the *<sequence>* is empty the result is empty. This is not yet available in XeTeX.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an *x*-type argument expansion.

<code>\seq_const_from_clist:Nn</code>	<code>\seq_const_from_clist:Nn <seq var> {<comma-list>}</code>
<code>\seq_const_from_clist:cn</code>	

New: 2017-11-28

Creates a new constant *<seq var>* or raises an error if the name is already taken. The *<seq var>* is set globally to contain the items in the *<comma list>*.

<code>\seq_set_from_function:NnN</code>	<code>\seq_set_from_function:NnN <seq var> {<loop code>} <function></code>
<code>\seq_gset_from_function:NnN</code>	

New: 2018-04-06

Sets the $\langle seq\ var \rangle$ equal to a sequence whose items are obtained by **x**-expanding $\langle loop\ code \rangle$ $\langle function \rangle$. This expansion must result in successive calls to the $\langle function \rangle$ with no nonexpandable tokens in between. More precisely the $\langle function \rangle$ is replaced by a wrapper function that inserts the appropriate separators between items in the sequence. The $\langle loop\ code \rangle$ must be expandable; it can be for example `\tl_map_function:NN <tl var>` or `\clist_map_function:nN {<clist>}` or `\int_step_function:nnnN {<initial value>} {<step>} {<final value>}`.

<code>\seq_set_from_inline_x:Nnn</code>	<code>\seq_set_from_inline_x:Nnn <seq var> {<loop code>} {<inline code>}</code>
<code>\seq_gset_from_inline_x:Nnn</code>	

New: 2018-04-06

Sets the $\langle seq\ var \rangle$ equal to a sequence whose items are obtained by **x**-expanding $\langle loop\ code \rangle$ applied to a $\langle function \rangle$ derived from the $\langle inline\ code \rangle$. A $\langle function \rangle$ is defined, that takes one argument, **x**-expands the $\langle inline\ code \rangle$ with that argument as **#1**, then adds appropriate separators to turn the result into an item of the sequence. The **x**-expansion of $\langle loop\ code \rangle$ $\langle function \rangle$ must result in successive calls to the $\langle function \rangle$ with no nonexpandable tokens in between. The $\langle loop\ code \rangle$ must be expandable; it can be for example `\tl_map_function:NN <tl var>` or `\clist_map_function:nN {<clist>}` or `\int_step_function:nnnN {<initial value>} {<step>} {<final value>}`, but not the analogous “inline” mappings.

<code>\seq_shuffle:N</code>	<code>\seq_shuffle:N <seq var></code>
<code>\seq_gshuffle:N</code>	

New: 2018-04-29

Sets the $\langle seq\ var \rangle$ to the result of placing the items of the $\langle seq\ var \rangle$ in a random order. Each item is (roughly) as likely to end up in any given position.

T_EXhackers note: For sequences with more than 13 items or so, only a small proportion of all possible permutations can be reached, because the random seed `\sys_rand_seed:` only has 28-bits. The use of `\toks` internally means that sequences with more than 32767 or 65535 items (depending on the engine) cannot be shuffled.

13 Additions to l3skip

<code>\skip_split_finite_else_action:nnNN</code>	<code>\skip_split_finite_else_action:nnNN {<skipexpr>} {<action>}</code>
	$\langle dimen_1 \rangle$ $\langle dimen_2 \rangle$

Checks if the $\langle skipexpr \rangle$ contains finite glue. If it does then it assigns $\langle dimen_1 \rangle$ the stretch component and $\langle dimen_2 \rangle$ the shrink component. If it contains infinite glue set $\langle dimen_1 \rangle$ and $\langle dimen_2 \rangle$ to 0pt and place **#2** into the input stream: this is usually an error or warning message of some sort.

14 Additions to l3sys

<hr/> <code>\sys_if_rand_exist_p: *</code> <code>\sys_if_rand_exist:TF *</code> <hr/> New: 2017-05-27	<code>\sys_if_rand_exist_p:</code> <code>\sys_if_rand_exist:TF {\true code}} {\false code}}</code> <p>Tests if the engine has a pseudo-random number generator. Currently this is the case in pdfTeX, LuaTeX, pTeX and upTeX.</p>
<hr/> <code>\sys_rand_seed: *</code> <hr/> New: 2017-05-27	<code>\sys_rand_seed:</code> <p>Expands to the current value of the engine's random seed, a non-negative integer. In engines without random number support this expands to 0.</p>
<hr/> <code>\sys_gset_rand_seed:n</code> <hr/> New: 2017-05-27	<code>\sys_gset_rand_seed:n {\intexpr}}</code> <p>Globally sets the seed for the engine's pseudo-random number generator to the <i>integer expression</i>. This random seed affects all <code>\..._rand</code> functions (such as <code>\int_rand:nn</code> or <code>\clist_rand_item:n</code>) as well as other packages relying on the engine's random number generator. In engines without random number support this produces an error.</p> <p>TeXhackers note: While a 32-bit (signed) integer can be given as a seed, only the absolute value is used and any number beyond 2^{28} is divided by an appropriate power of 2. We recommend using an integer in $[0, 2^{28} - 1]$.</p>
<hr/> <code>\c_sys_shell_escape_int</code> <hr/> New: 2017-05-27	<p>This variable exposes the internal triple of the shell escape status. The possible values are</p> <p>0 Shell escape is disabled</p> <p>1 Unrestricted shell escape is enabled</p> <p>2 Restricted shell escape is enabled</p>
<hr/> <code>\sys_if_shell_p: *</code> <code>\sys_if_shell:TF *</code> <hr/> New: 2017-05-27	<code>\sys_if_shell_p:</code> <code>\sys_if_shell:TF {\true code}} {\false code}}</code> <p>Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.</p>
<hr/> <code>\sys_if_shell_unrestricted_p: *</code> <code>\sys_if_shell_unrestricted:TF *</code> <hr/> New: 2017-05-27	<code>\sys_if_shell_unrestricted_p:</code> <code>\sys_if_shell_unrestricted:TF {\true code}} {\false code}}</code> <p>Performs a check for whether <i>unrestricted</i> shell escape is enabled.</p>
<hr/> <code>\sys_if_shell_restricted_p: *</code> <code>\sys_if_shell_restricted:TF *</code> <hr/> New: 2017-05-27	<code>\sys_if_shell_restricted_p:</code> <code>\sys_if_shell_restricted:TF {\true code}} {\false code}}</code> <p>Performs a check for whether <i>restricted</i> shell escape is enabled. This returns false if unrestricted shell escape is enabled. Unrestricted shell escape is not considered a superset of restricted shell escape in this case. To find whether any shell escape is enabled use <code>\sys_if_shell:.</code></p>

<code>\sys_shell_now:n</code>	<code>\sys_shell_now:n {⟨tokens⟩}</code>
<code>\sys_shell_now:x</code>	Execute <i>⟨tokens⟩</i> through shell escape immediately.

New: 2017-05-27

<code>\sys_shell_shipout:n</code>	<code>\sys_shell_shipout:n {⟨tokens⟩}</code>
<code>\sys_shell_shipout:x</code>	Execute <i>⟨tokens⟩</i> through shell escape at shipout.

New: 2017-05-27

15 Additions to l3tl

<code>\tl_if_single_token_p:n</code> ★	<code>\tl_if_single_token_p:n {⟨token list⟩}</code>
<code>\tl_if_single_token:nTF</code> ★	<code>\tl_if_single_token:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>

Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single “normal” token. Token groups (`{...}`) are not single tokens.

<code>\tl_reverse_tokens:n</code> ★	<code>\tl_reverse_tokens:n {⟨tokens⟩}</code>
-------------------------------------	--

This function, which works directly on T_EX tokens, reverses the order of the *⟨tokens⟩*: the first becomes the last and the last becomes first. Spaces are preserved. The reversal also operates within brace groups, but the braces themselves are not exchanged, as this would lead to an unbalanced token list. For instance, `\tl_reverse_tokens:n {a~{b{}}}` leaves `{ } (b) ~a` in the input stream. This function requires two steps of expansion.

T_EXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the token list does not expand further when appearing in an x-type argument expansion.

<code>\tl_count_tokens:n</code> ★	<code>\tl_count_tokens:n {⟨tokens⟩}</code>
-----------------------------------	--

Counts the number of T_EX tokens in the *⟨tokens⟩* and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of `a~{bc}` is 6. This function requires three expansions, giving an *⟨integer denotation⟩*.

<code>\tl_lower_case:n</code> ★	<code>\tl_upper_case:n {⟨tokens⟩}</code>
<code>\tl_upper_case:n</code> ★	<code>\tl_upper_case:nn {⟨language⟩} {⟨tokens⟩}</code>
<code>\tl_mixed_case:n</code> ★	
<code>\tl_lower_case:nn</code> ★	
<code>\tl_upper_case:nn</code> ★	
<code>\tl_mixed_case:nn</code> ★	

New: 2014-06-30
Updated: 2016-01-12

These functions are intended to be applied to input which may be regarded broadly as “text”. They traverse the *⟨tokens⟩* and change the case of characters as discussed below. The character code of the characters replaced may be arbitrary: the replacement characters have standard document-level category codes (11 for letters, 12 for letter-like characters which can also be case-changed). Begin-group and end-group characters in the *⟨tokens⟩* are normalized and become `{` and `}`, respectively.

Importantly, notice that these functions are intended for working with user text for typesetting. For case changing programmatic data see the `l3str` module and discussion there of `\str_lower_case:n`, `\str_upper_case:n` and `\str_fold_case:n`.

The functions perform expansion on the input in most cases. In particular, input in the form of token lists or expandable functions is expanded *unless* it falls within one

of the special handling classes described below. This expansion approach means that in general the result of case changing matches the “natural” outcome expected from a “functional” approach to case modification. For example

```
\tl_set:Nn \l_tmpa_tl { hello }
\tl_upper_case:n { \l_tmpa_tl \c_space_tl world }
```

produces

```
HELLO WORLD
```

The expansion approach taken means that in package mode any L^AT_EX 2_ε “robust” commands which may appear in the input should be converted to engine-protected versions using for example the `\robustify` command from the `etoolbox` package.

`\l_tl_case_change_math_tl`

Case changing does not take place within math mode material so for example

```
\tl_upper_case:n { Some~text~$y = mx + c$~with~{Braces} }
```

becomes

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

Material inside math mode is left entirely unchanged: in particular, no expansion is undertaken.

Detection of math mode is controlled by the list of tokens in `\l_tl_case_change_math_tl`, which should be in open-close pairs. In package mode the standard settings is

```
$ $ \ ( \)
```

Note that while expansion occurs when searching the text it does not apply to math mode material (which should be unaffected by case changing). As such, whilst the opening token for math mode may be “hidden” inside a command/macro, the closing one cannot be as this is being searched for in math mode. Typically, in the types of “text” the case changing functions are intended to apply to this should not be an issue.

`\l_tl_case_change_exclude_tl`

Case changing can be prevented by using any command on the list `\l_tl_case_change_exclude_tl`. Each entry should be a function to be followed by one argument: the latter will be preserved as-is with no expansion. Thus for example following

```
\tl_put_right:Nn \l_tl_case_change_exclude_tl { \NoChangeCase }
```

the input

```
\tl_upper_case:n  
  { Some~text~$y = mx + c$~with~\NoChangeCase {Protection} }
```

will result in

```
SOME TEXT $y = mx + c$ WITH \NoChangeCase {Protection}
```

Notice that the case changing mapping preserves the inclusion of the escape functions: it is left to other code to provide suitable definitions (typically equivalent to `\use:n`). In particular, the result of case changing is returned protected by `\exp_not:n`.

When used with L^AT_EX 2_ε the commands `\cite`, `\ensuremath`, `\label` and `\ref` are automatically included in the list for exclusion from case changing.

`\l_tl_case_change_accents_tl`

This list specifies accent commands which should be left unexpanded in the output. This allows for example

```
\tl_upper_case:n { \" { a } }
```

to yield

```
\" { A }
```

irrespective of the expandability of `\`.

The standard contents of this variable is `\`, `\'`, `\.`, `\^`, `\'`, `\~`, `\c`, `\H`, `\k`, `\r`, `\t`, `\u` and `\v`.

“Mixed” case conversion may be regarded informally as converting the first character of the *<tokens>* to upper case and the rest to lower case. However, the process is more complex than this as there are some situations where a single lower case character maps to a special form, for example *ij* in Dutch which becomes *IJ*. As such, `\tl_mixed_case:n(n)` implement a more sophisticated mapping which accounts for this and for modifying accents on the first letter. Spaces at the start of the *<tokens>* are ignored when finding the first “letter” for conversion.

```
\tl_mixed_case:n { hello~WORLD } % => "Hello world"  
\tl_mixed_case:n { ~hello~WORLD } % => " Hello world"  
\tl_mixed_case:n { {hello}~WORLD } % => "{Hello} world"
```

When finding the first “letter” for this process, any content in math mode or covered by `\l_tl_case_change_exclude_tl` is ignored.

(Note that the Unicode Consortium describe this as “title case”, but that in English title case applies on a word-by-word basis. The “mixed” case implemented here is a lower level concept needed for both “title” and “sentence” casing of text.)

`\tl_mixed_case_ignore_tl`

The list of characters to ignore when searching for the first “letter” in mixed-casing is determined by `\tl_mixed_change_ignore_tl`. This has the standard setting

`([{ ‘ -`

where comparisons are made on a character basis.

As is generally true for `expl3`, these functions are designed to work with Unicode input only. As such, UTF-8 input is assumed for *all* engines. When used with `XYTeX` or `LuaTeX` a full range of Unicode transformations are enabled. Specifically, the standard mappings here follow those defined by the [Unicode Consortium](#) in `UnicodeData.txt` and `SpecialCasing.txt`. In the case of 8-bit engines, mappings are provided for characters which can be represented in output typeset using the T1 font encoding. Thus for example ä can be case-changed using `pdfTeX`. For `pTeX` only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Context-sensitive mappings are enabled: language-dependent cases are discussed below. Context detection expands input but treats any unexpandable control sequences as “failures” to match a context.

Language-sensitive conversions are enabled using the `<language>` argument, and follow Unicode Consortium guidelines. Currently, the languages recognised for special handling are as follows.

- Azeri and Turkish (`az` and `tr`). The case pairs I/i-dotless and I-dot/i are activated for these languages. The combining dot mark is removed when lower casing I-dot and introduced when upper casing i-dotless.
- German (`de-alt`). An alternative mapping for German in which the lower case *Eszett* maps to a *großes Eszett*.
- Lithuanian (`lt`). The lower case letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lower casing of the relevant upper case letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when upper casing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.
- Dutch (`nl`). Capitalisation of `ij` at the beginning of mixed cased input produces `IJ` rather than `Ij`. The output retains two separate letters, thus this transformation *is* available using `pdfTeX`.

Creating additional context-sensitive mappings requires knowledge of the underlying mapping implementation used here. The team are happy to add these to the kernel where they are well-documented (*e.g.* in Unicode Consortium or relevant government publications).

`\tl_set_from_file:Nnn`
`\tl_set_from_file:cnn`
`\tl_gset_from_file:Nnn`
`\tl_gset_from_file:cnn`

New: 2014-06-25

`\tl_set_from_file:Nnn <tl> {<setup>} {<filename>}`

Defines `<tl>` to the contents of `<filename>`. Category codes may need to be set appropriately via the `<setup>` argument.

<code>\tl_set_from_file_x:Nnn</code>	<code>\tl_set_from_file_x:Nnn <tl> {<setup>} {<filename>}</code>
<code>\tl_set_from_file_x:cnn</code>	Defines <code><tl></code> to the contents of <code><filename></code> , expanding the contents of the file as it is read.
<code>\tl_gset_from_file_x:Nnn</code>	Category codes and other definitions may need to be set appropriately via the <code><setup></code>
<code>\tl_gset_from_file_x:cnn</code>	argument.

New: 2014-06-25

<code>\tl_rand_item:N *</code>	<code>\tl_rand_item:N <tl var></code>
<code>\tl_rand_item:c *</code>	<code>\tl_rand_item:n {<token list>}</code>
<code>\tl_rand_item:n *</code>	Selects a pseudo-random item of the <code><token list></code> . If the <code><token list></code> is blank, the result is empty. This is not yet available in Xe _{La} TeX.

New: 2016-12-06

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<item>` does not expand further when appearing in an `x`-type argument expansion.

<code>\tl_range:Nnn *</code>	<code>\tl_range:Nnn <tl var> {<start index>} {<end index>}</code>
<code>\tl_range:nnn *</code>	<code>\tl_range:nnn {<token list>} {<start index>} {<end index>}</code>

New: 2017-02-17

Updated: 2017-07-15

Leaves in the input stream the items from the `<start index>` to the `<end index>` inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Positive `<indices>` are counted from the start of the `<token list>`, 1 being the first item, and negative `<indices>` are counted from the end of the token list, `-1` being the last item. If either of `<start index>` or `<end index>` is 0, the result is empty. For instance,

```
\iow_term:x { \tl_range:nnn { abcd~{e}}f } { 2 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd~{e}}f } { -4 } { -1 } }
\iow_term:x { \tl_range:nnn { abcd~{e}}f } { -2 } { -1 } }
\iow_term:x { \tl_range:nnn { abcd~{e}}f } { 0 } { -1 } }
```

prints `bcd_{e}`, `cd_{e}f`, `{e}f` and an empty line to the terminal. The `<start index>` must always be smaller than or equal to the `<end index>`: if this is not the case then no output is generated. Thus

```
\iow_term:x { \tl_range:nnn { abcd~{e}}f } { 5 } { 2 } }
\iow_term:x { \tl_range:nnn { abcd~{e}}f } { -1 } { -4 } }
```

both yield empty token lists. For improved performance, see `\tl_range_braced:nnn` and `\tl_range_unbraced:nnn`.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<item>` does not expand further when appearing in an `x`-type argument expansion.

<code>\tl_range_braced:Nnn</code>	★	<code>\tl_range_braced:Nnn <tl var> {<start index>} {<end index>}</code>
<code>\tl_range_braced:cnn</code>	★	<code>\tl_range_braced:nnn {<token list>} {<start index>} {<end index>}</code>
<code>\tl_range_braced:nnn</code>	★	<code>\tl_range_unbraced:Nnn <tl var> {<start index>} {<end index>}</code>
<code>\tl_range_unbraced:Nnn</code>	★	<code>\tl_range_unbraced:nnn {<token list>} {<start index>} {<end index>}</code>
<code>\tl_range_unbraced:cnn</code>	★	Leaves in the input stream the items from the <code><start index></code> to the <code><end index></code> inclusive, using the same indexing as <code>\tl_range:nnn</code> . Spaces are ignored. Regardless of whether items appear with or without braces in the <code><token list></code> , the <code>\tl_range_braced:nnn</code> function wraps each item in braces, while <code>\tl_range_unbraced:nnn</code> does not (overall it removes an outer set of braces). For instance,
<code>\tl_range_unbraced:nnn</code>	★	

New: 2017-07-15

```

\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { 2 } { 5 } }
\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { -4 } { -1 } }
\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { -2 } { -1 } }
\tl_iow_term:x { \tl_range_braced:nnn { abcd~{e}}f } { 0 } { -1 } }

```

prints `{b}{c}{d}{e}`, `{c}{d}{e}{f}`, `{e}{f}`, and an empty line to the terminal, while

```

\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { 2 } { 5 } }
\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { -4 } { -1 } }
\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { -2 } { -1 } }
\tl_iow_term:x { \tl_range_unbraced:nnn { abcd~{e}}f } { 0 } { -1 } }

```

prints `bcde{}`, `cde{f}`, `e{f}`, and an empty line to the terminal. Because braces are removed, the result of `\tl_range_unbraced:nnn` may have a different number of items as for `\tl_range:nnn` or `\tl_range_braced:nnn`. In cases where preserving spaces is important, consider the slower function `\tl_range:nnn`.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<item>` does not expand further when appearing in an `x`-type argument expansion.

<code>\tl_build_begin:N</code>	<code>\tl_build_begin:N <tl var></code>
<code>\tl_build_gbegin:N</code>	Clears the <code><tl var></code> and sets it up to support other <code>\tl_build_...</code> functions, which allow accumulating large numbers of tokens piece by piece much more efficiently than standard <code>\l3tl</code> functions. Until <code>\tl_build_end:N <tl var></code> is called, applying any function from <code>\l3tl</code> other than <code>\tl_build_...</code> will lead to incorrect results. The <code>begin</code> and <code>gbegin</code> functions must be used for local and global <code><tl var></code> respectively.
New: 2018-04-01	

<code>\tl_build_clear:N</code>	<code>\tl_build_clear:N <tl var></code>
<code>\tl_build_gclear:N</code>	Clears the <code><tl var></code> and sets it up to support other <code>\tl_build_...</code> functions. The <code>clear</code> and <code>gclear</code> functions must be used for local and global <code><tl var></code> respectively.
New: 2018-04-01	

```

\tl_build_put_left:Nn
\tl_build_put_left:Nx
\tl_build_gput_left:Nn
\tl_build_gput_left:Nx
\tl_build_put_right:Nn
\tl_build_put_right:Nx
\tl_build_gput_right:Nn
\tl_build_gput_right:Nx

```

New: 2018-04-01

```

\tl_build_get:NN

```

New: 2018-04-01

```

\tl_build_end:N
\tl_build_gend:N

```

New: 2018-04-01

```

\tl_build_put_left:Nn <tl var> {<tokens>}
\tl_build_put_right:Nn <tl var> {<tokens>}

```

Adds *<tokens>* to the left or right side of the current contents of *<tl var>*. The *<tl var>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `put` and `gput` functions must be used for local and global *<tl var>* respectively. The `right` functions are about twice faster than the `left` functions.

```

\tl_build_get:N <tl var1> <tl var2>

```

Stores the contents of the *<tl var₁>* in the *<tl var₂>*. The *<tl var₁>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The *<tl var₂>* is a “normal” token list variable, assigned locally using `\tl_set:Nn`.

```

\tl_build_end:N <tl var>

```

Gets the contents of *<tl var>* and stores that into the *<tl var>* using `\tl_set:Nn`. The *<tl var>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `end` and `gend` functions must be used for local and global *<tl var>* respectively. These functions completely remove the setup code that enabled *<tl var>* to be used for other `\tl_build_...` functions.

16 Additions to l3token

```

\c_catcode_active_space_tl

```

New: 2017-08-07

Token list containing one character with category code 13, (“active”), and character code 32 (space).

```

\char_lower_case:N ★
\char_upper_case:N ★
\char_mixed_case:N ★
\char_fold_case:N ★

```

New: 2018-04-06

```

\char_lower_case:N <char>

```

Converts the *<char>* to the equivalent case-changed character as detailed by the function name (see `\str_fold_case:n` and `\tl_mixed_case:n` for details of these terms). The case mapping is carried out with no context-dependence (*cf.* `\tl_upper_case:n`, *etc.*)

```

\char_codepoint_to_bytes:n

```

New: 2018-04-02

```

\char_codepoint_to_bytes:n {<codepoint>}

```

Converts the (Unicode) *<codepoint>* to UTF-8 bytes. The expansion of this function comprises a leading integer (1–4) followed by the appropriate number of brace groups, each of which contains one byte of the UTF-8 sequence for the codepoint.

`\peek_N_type:TF`

Updated: 2012-12-20

`\peek_N_type:TF {<true code>} {<false code>}`

Tests if the next *<token>* in the input stream can be safely grabbed as an N-type argument. The test is *<false>* if the next *<token>* is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in L^AT_EX3) and *<true>* in all other cases. Note that a *<true>* result ensures that the next *<token>* is a valid N-type argument. However, if the next *<token>* is for instance `\c_space_token`, the test takes the *<false>* branch, even though the next *<token>* is in fact a valid N-type argument. The *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

Part XXXII

The l3drivers package

Drivers

T_EX relies on drivers in order to carry out a number of tasks, such as using color, including graphics and setting up hyper-links. The nature of the code required depends on the exact driver in use. Currently, L^AT_EX3 is aware of the following drivers:

- **pdfmode**: The “driver” for direct PDF output by *both* pdfT_EX and LuaT_EX (no separate driver is used in this case: the engine deals with PDF creation itself).
- **dvips**: The dvips program, which works in conjugation with pdfT_EX or LuaT_EX in DVI mode.
- **dvipdfmx**: The dvipdfmx program, which works in conjugation with pdfT_EX or LuaT_EX in DVI mode.
- **dvisvgm**: The dvisvgm program, which works in conjugation with pdfT_EX or LuaT_EX when run in DVI mode as well as with (u)pT_EX and X_YT_EX.
- **xdvipdfmx**: The driver used by X_YT_EX.

This module provides code closely tied to the exact driver in use: broadly, the functions here are implemented entirely independently for each case. As such, they often rely on higher-level code to provide necessary but shared operations. For example, in box rotation and scaling the functions here do not correct the final size of the box: this will always be required and thus is handled in the `box` module.

Several of the operations here are low-level, and so may be used only in restricted contexts. Some also require understanding of PostScript/PDF concepts to be used correctly as they take “raw” arguments, similar in format to those used by the underlying driver.

The functions in this module should be regarded as experimental with the following exceptions:

- ...

1 Box clipping

`\driver_box_use_clip:N`

New: 2017-12-13

`\driver_box_use_clip:N` $\langle box \rangle$

Inserts the content of the $\langle box \rangle$ at the current insertion point such that any material outside of the bounding box is not displayed by the driver. The material in the $\langle box \rangle$ is still placed in the output stream: the clipping takes place at a driver level.

2 Box rotation and scaling

`\driver_box_use_rotate:Nn`

New: 2017-12-13
Updated: 2018-04-26

`\driver_box_use_rotate:Nn <box> {<angle>}`

Inserts the content of the `<box>` at the current insertion point rotated by the `<angle>` (an *fp expression*) expressed in degrees). The material is rotated such the the $\mathrm{T\!E\!X}$ reference point of the box is the center of rotation and remains the reference point after rotation. It is the responsibility of the code using this function to adjust the apparent size of the inserted material.

`\driver_box_use_scale:Nnn`

New: 2017-12-13
Updated: 2018-04-26

`\driver_box_use_scale:Nnn <box> {<x-scale>} {<y-scale>}`

Inserts the content of the `<box>` at the current insertion point scale by the `<x-scale>` and `<y-scale>` (both *fp expressions*). The reference point of the material will be unchanged. It is the responsibility of the code using this function to adjust the apparent size of the inserted material.

3 Color support

`\driver_color_cmyk:nnnn`

New: 2018-02-20

`\driver_color_cmyk:nnnn {<cyan>} {<magenta>} {<yellow>} {<black>}`

Sets the color to the CMYK values specified, all of which are fp denotations in the range 0 and 1. For drawing colors, see `\driver_draw_stroke_cmyk:nnnn`, *etc.*

`\driver_color_gray:n`

New: 2018-02-20

`\driver_color_gray:n {<gray>}`

Sets the color to the grayscale value specified, which is fp denotations in the range 0 and 1. For drawing colors, see `\driver_draw_stroke_gray:n`, *etc.*

`\driver_color_rgb:nnn`

New: 2018-02-20

`\driver_color_rgb:nnn {<red>} {<green>} {<blue>}`

Sets the color to the RGB values specified, all of which are fp denotations in the range 0 and 1. For drawing colors, see `\driver_draw_stroke_rgb:nnn`, *etc.*

`\driver_color_pickup:N`

New: 2018-02-20

`\driver_color_pickup:N <tl>`

In $\mathrm{L\!A\!T\!E\!X}\ 2_{\epsilon}$ package mode, collects data on the current color from `\current@color` and stores it in the low-level format used by `expl3` in the `<tl>`.

4 Drawing

The drawing functions provided here are *highly* experimental. They are inspired heavily by the system layer of `pgf` (most have the same interface as the same functions in the latter's `\pgfsys@...` namespace). They are intended to form the basis for higher level drawing interfaces, which themselves are likely to be further abstracted for user access. Again, this model is heavily inspired by `pgf` and `Tikz`.

These low level drawing interfaces abstract from the driver raw requirements but still require an appreciation of the concepts of PostScript/PDF/SVG graphic creation.

<hr/> <hr/>	<code>\driver_draw_begin:</code>
<code>\driver_draw_end:</code>	<code>\driver_draw_end:</code>
	<code>\driver_draw_end:</code>
	Defines a drawing environment. This is a scope for the purposes of the graphics state. Depending on the driver, other set up may or may not take place here. The natural size of the <code>\content</code> should be zero from the T _E X perspective: allowance for the size of the content must be made at a higher level (or indeed this can be skipped if the content is to overlap other material).

<hr/> <hr/>	<code>\driver_draw_scope_begin:</code>
<code>\driver_draw_scope_end:</code>	<code>\driver_draw_scope_end:</code>
	<code>\driver_draw_scope_end:</code>
	Defines a scope for drawing settings and so on. Changes to the graphic state and concepts such as color or linewidth are localised to a scope. This function pair must never be used if an partial path is under construction: such paths must be entirely contained at one unbroken scope level. Note that scopes do not form T _E X groups and may not be aligned with them.

4.1 Path construction

<hr/> <hr/>	<code>\driver_draw_moveto:nn</code>
	<code>\driver_draw_move:nn {<x>} {<y>}</code>
	Moves the current drawing reference point to $(\langle x \rangle, \langle y \rangle)$; any active transformation matrix applies.

<hr/> <hr/>	<code>\driver_draw_lineto:nn</code>
	<code>\driver_draw_lineto:nn {<x>} {<y>}</code>
	Adds a path from the current drawing reference point to $(\langle x \rangle, \langle y \rangle)$; any active transformation matrix applies. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

<hr/> <hr/>	<code>\driver_draw_curveto:nnnnnn</code>
	<code>\driver_draw_curveto:nnnnnn {<x₁>} {<y₁>} {<x₂>} {<y₂>} {<x₃>} {<y₃>}</code>
	Adds a Bezier curve path from the current drawing reference point to $(\langle x_3 \rangle, \langle y_3 \rangle)$, using $(\langle x_1 \rangle, \langle y_1 \rangle)$ and $(\langle x_2 \rangle, \langle y_2 \rangle)$ as control points; any active transformation matrix applies. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

<hr/> <hr/>	<code>\driver_draw_rectangle:nnnn</code>
	<code>\driver_draw_rectangle:nnnn {<x>} {<y>} {<width>} {<height>}</code>
	Adds rectangular path from $(\langle x_1 \rangle, \langle y_1 \rangle)$ of $\langle height \rangle$ and $\langle width \rangle$; any active transformation matrix applies. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

<hr/> <hr/>	<code>\driver_draw_closepath:</code>
	<code>\driver_draw_closepath:</code>
	Closes an existing path, adding a line from the current point to the start of path. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

4.2 Stroking and filling

<code>\driver_draw_stroke:</code>	<code><path construction></code>
<code>\driver_draw_closestroke:</code>	<code>\driver_draw_stroke:</code>

Draws a line along the current path, which is also closed in the case of `\driver_draw_closestroke:`. The nature of the line drawn is influenced by settings for

- Line thickness
- Stroke color (or the current color if no specific stroke color is set)
- Line capping (how non-closed line ends should look)
- Join style (how a bend in the path should be rendered)
- Dash pattern

The path may also be used for clipping.

<code>\driver_draw_fill:</code>	<code><path construction></code>
<code>\driver_draw_fillstroke:</code>	<code>\driver_draw_fill:</code>

Fills the area surrounded by the current path: this will be closed prior to filling if it is not already. The `fillstroke` version also strokes the path as described for `\driver_draw_stroke:`. The fill is influenced by the setting for fill color (or the current color if no specific stroke color is set). The path may also be used for clipping. For paths which are self-intersecting or comprising multiple parts, the determination of which areas are inside the path is made using the non-zero winding number rule unless the even-odd rule is active.

<code>\driver_draw_nonzero_rule:</code>	<code>\driver_draw_nonzero_rule:</code>
<code>\driver_draw_evenodd_rule:</code>	

Active either the non-zero winding number or the even-odd rule, respectively, for determining what is inside a fill or clip area. For technical reasons, these command are not influenced by scoping and apply on an ongoing basis.

<code>\driver_draw_clip:</code>	<code><path construction></code>
	<code>\driver_draw_clip:</code>

Indicates that the current path should be used for clipping, such that any subsequent material outside of the path (but within the current scope) will not be shown. This command should be given once a path is complete but before it is stroked or filled (if appropriate). This command is *not* affected by scoping: it applies to exactly one path as shown.

<code>\driver_draw_discardpath:</code>	<code><path construction></code>
	<code>\driver_draw_discardpath:</code>

Discards the current path without stroking or filling. This is primarily useful for paths constructed purely for clipping, as this alone does not end the paths existence.

4.3 Stroke options

<code>\driver_draw_linewidth:n</code>	<code>\driver_draw_linewidth:n {<dimexpr>}</code>
---------------------------------------	---

Sets the width to be used for stroking to *<dimexpr>*.

<code>\driver_draw_dash_pattern:nn</code>	<code>\driver_draw_dash:nn {<dash pattern>} {<phase>}</code>
---	--

Sets the pattern of dashing to be used when stroking a line. The *<dash pattern>* should be a comma-separated list of dimension expressions. This is then interpreted as a series of pairs of line-on and line-off lengths. For example `3pt, 4pt` means that 3pt on, 4pt off, 3pt on, and so on. A more complex pattern will also repeat: `3pt, 4pt, 1pt, 2pt` results in 3pt on, 4pt off, 1pt on, 2pt off, 3pt on, and so on. An odd number of entries means that the last is repeated, for example `3pt` is equal to `3pt, 3pt`. An empty pattern yields a solid line.

The *<phase>* specifies an offset at the start of the cycle. For example, with a pattern `3pt` a phase of `1pt` means that the output is 2pt on, 3pt off, 3pt on, 3pt on, *etc.*

<code>\driver_draw_cap_but:</code>	<code>\driver_draw_cap_but:</code>
<code>\driver_draw_cap_rectangle:</code>	
<code>\driver_draw_cap_round:</code>	

Sets the style of terminal stroke position to one of butt, rectangle or round.

<code>\driver_draw_join_bevel:</code>	<code>\driver_draw_cap_but:</code>
<code>\driver_draw_join_miter:</code>	
<code>\driver_draw_join_round:</code>	

Sets the style of stroke joins to one of bevel, miter or round.

<code>\driver_draw_miterlimit:n</code>	<code>\driver_draw_miterlimit:n {<factor>}</code>
--	---

Sets the miter limit of lines joined as a miter, as described in the PDF and PostScript manuals. The *<factor>* here is an *<fp expression>*.

4.4 Color

<code>\driver_draw_color_fill_cmyk:nnnn</code>	<code>\driver_draw_color_fill_cmyk:nnnn {<cyan>} {<magenta>} {<yellow>}</code>
<code>\driver_draw_color_stroke_cmyk:nnnn</code>	<code>{<black>}</code>

Sets the color for drawing to the CMYK values specified, all of which are fp denotations in the range 0 and 1.

<code>\driver_draw_color_fill_gray:n</code>	<code>\driver_draw_color_fill_gray:n {<gray>}</code>
<code>\driver_draw_color_stroke_gray:n</code>	

Sets the color for drawing to the grayscale value specified, which is fp denotations in the range 0 and 1.

<code>\driver_draw_color_fill_rgb:nnn</code>	<code>\driver_draw_color_fill_rgb:nnn {<red>} {<green>} {<blue>}</code>
<code>\driver_draw_color_stroke_rgb:nnn</code>	

Sets the color for drawing to the RGB values specified, all of which are fp denotations in the range 0 and 1.

4.5 Inserting T_EX material

```
\driver_draw_box_use:Nnnnn
```

```
\driver_draw_box:Nnnnnnnn <box>
{\<a>} {\<b>} {\<c>} {\<d>} {\<x>} {\<y>}
```

Inserts the $\langle box \rangle$ as an hbox with the box reference point placed at (x, y) . The transformation matrix $[abcd]$ is applied to the box, allowing it to be in synchronisation with any scaling, rotation or skewing applying more generally. Note that T_EX material should not be inserted directly into a drawing as it would not be in the correct location. Also note that as for other drawing elements the box here has no size from a T_EX perspective.

4.6 Coordinate system transformations

```
\driver_draw_cm:nnnn
```

```
\driver_draw_cm:nnnn {\<a>} {\<b>} {\<c>} {\<d>}
```

Applies the transformation matrix $[abcd]$ to the current graphic state. This affects any subsequent items in the same scope but not those already given.

Part XXXIII

Implementation

1 l3bootstrap implementation

```
1 <*initex | package>
2 <@@=kernel>
```

1.1 Format-specific code

The very first thing to do is to bootstrap the iniT_EX system so that everything else will actually work. T_EX does not start with some pretty basic character codes set up.

```
3 <*initex>
4 \catcode '\{ = 1 %
5 \catcode '\} = 2 %
6 \catcode '\# = 6 %
7 \catcode '\^ = 7 %
8 </initex>
```

Tab characters should not show up in the code, but to be on the safe side.

```
9 <*initex>
10 \catcode '\^^I = 10 %
11 </initex>
```

For LuaT_EX, the extra primitives need to be enabled. This is not needed in package mode: common formats have the primitives enabled.

```
12 <*initex>
13 \begingroup\expandafter\expandafter\expandafter\endgroup
14 \expandafter\ifx\csgname directlua\endcsgname\relax
15 \else
16 \directlua{tex.enableprimitives("", tex.extraprimitives())}%
17 \fi
18 </initex>
```

Depending on the versions available, the L^AT_EX format may not have the raw `\Umath` primitive names available. We fix that globally: it should cause no issues. Older LuaT_EX versions do not have a pre-built table of the primitive names here so sort one out ourselves. These end up globally-defined but at that is true with a newer format anyway and as they all start `\U` this should be reasonably safe.

```

19 \begin{package}
20 \begin{group}
21   \expandafter\ifx\csname directlua\endcsname\relax
22   \else
23     \directlua{%
24       local i
25       local t = { }
26       for _,i in pairs(tex.extraprimitives("luatex")) do
27         if string.match(i,"^U") then
28           if not string.match(i,"^Uchar$") then %$
29             table.insert(t,i)
30           end
31         end
32       end
33       tex.enableprimitives("", t)
34     }%
35   \fi
36 \end{group}
37 \end{package}

```

1.2 The `\pdfstrcmp` primitive in X_YT_EX

Only pdfT_EX has a primitive called `\pdfstrcmp`. The X_YT_EX version is just `\strcmp`, so there is some shuffling to do. As this is still a real primitive, using the pdfT_EX name is “safe”.

```

38 \begin{group}\expandafter\expandafter\expandafter\end{group}
39 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
40   \let\pdfstrcmp\strcmp
41 \fi

```

1.3 Loading support Lua code

When LuaT_EX is used there are various pieces of Lua code which need to be loaded. The code itself is defined in `l3luatex` and is extracted into a separate file. Thus here the task is to load the Lua code both now and (if required) at the start of each job.

```

42 \begin{group}\expandafter\expandafter\expandafter\end{group}
43 \expandafter\ifx\csname directlua\endcsname\relax
44 \else
45   \ifnum\luatexversion<70 %
46   \else

```

In package mode a category code table is needed: either use a pre-loaded allocator or provide one using the L^AT_EX 2_ε-based generic code. In format mode the table used here can be hard-coded into the Lua.

```

47 \begin{package}
48   \begin{group}\expandafter\expandafter\expandafter\end{group}
49   \expandafter\ifx\csname newcatcodetable\endcsname\relax

```

```

50     \input{ltluatex}%
51 \fi
52 \newcatcodetable\ucharcat@table
53 \directlua{
54     l3kernel = l3kernel or { }
55     local charcat_table = \number\ucharcat@table\space
56     l3kernel.charcat_table = charcat_table
57 }%
58 \endpackage
59 \directlua{require("expl3")}%

```

As the user might be making a custom format, no assumption is made about matching package mode with only loading the Lua code once. Instead, a query to Lua reveals what mode is in operation.

```

60 \ifnum 0%
61 \directlua{
62     if status.ini_version then
63         tex.write("1")
64     end
65 }>0 %
66 \everyjob\expandafter{%
67     \the\expandafter\everyjob
68     \csname\detokenize{lua_now_x:n}\endcsname{require("expl3")}%
69 }%
70 \fi
71 \fi
72 \fi

```

1.4 Engine requirements

The code currently requires ϵ -TeX and functionality equivalent to `\pdfstrcmp`, and also driver and Unicode character support. This is available in a reasonably-wide range of engines.

```

73 \begingroup
74 \def\next{\endgroup}%
75 \def\ShortText{Required primitives not found}%
76 \def\LongText%
77 {%
78     LaTeX3 requires the e-TeX primitives and additional functionality as
79     described in the README file.
80     \LineBreak
81     These are available in the engines\LineBreak
82     - pdfTeX v1.40\LineBreak
83     - XeTeX v0.9994\LineBreak
84     - LuaTeX v0.70\LineBreak
85     - e-(u)pTeX mid-2012\LineBreak
86     or later.\LineBreak
87     \LineBreak
88 }%
89 \ifnum0%
90 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
91 \else
92 \expandafter\ifx\csname pdftexversion\endcsname\relax
93     1%

```

```

94     \else
95       \ifnum\pdfTeXversion<140 \else 1\fi
96     \fi
97   \fi
98   \expandafter\ifx\csname directlua\endcsname\relax
99   \else
100     \ifnum\luaTeXversion<70 \else 1\fi
101   \fi
102   =0 %
103   \newlinechar'\^^J %
104   <*\initex>
105   \def\LineBreak{^^J}%
106   \edef\next
107     {%
108     \errhelp
109     {%
110       \LongText
111       For pdfTeX and XeTeX the '-etex' command-line switch is also
112       needed.\LineBreak
113       \LineBreak
114       Format building will abort!\LineBreak
115     }%
116     \errmessage{\ShortText}%
117     \endgroup
118     \noexpand\end
119   }%
120   </\initex>
121   <*\package>
122     \def\LineBreak{\noexpand\MessageBreak}%
123     \expandafter\ifx\csname PackageError\endcsname\relax
124       \def\LineBreak{^^J}%
125       \def\PackageError#1#2#3%
126         {%
127           \errhelp{#3}%
128           \errmessage{#1 Error: #2}%
129         }%
130     \fi
131     \edef\next
132       {%
133       \noexpand\PackageError{expl3}{\ShortText}
134       {\LongText Loading of expl3 will abort!}%
135       \endgroup
136       \noexpand\endinput
137     }%
138   </\package>
139   \fi
140   \next

```

1.5 Extending allocators

In format mode, allocating registers is handled by `l3alloc`. However, in package mode it's much safer to rely on more general code. For example, the ability to extend \TeX 's allocation routine to allow for $\varepsilon\text{-}\TeX$ has been around since 1997 in the `etex` package.

Loading this support is delayed until here as we are now sure that the ε -TeX extensions and `\pdfstrcmp` or equivalent are available. Thus there is no danger of an “uncontrolled” error if the engine requirements are not met.

For L^AT_EX_ε we need to make sure that the extended pool is being used: `expl3` uses a lot of registers. For formats from 2015 onward there is nothing to do as this is automatic. For older formats, the `etex` package needs to be loaded to do the job. In that case, some inserts are reserved also as these have to be from the standard pool. Note that `\reserveinserts` is `\outer` and so is accessed here by `csname`. In earlier versions, loading `etex` was done directly and so `\reserveinserts` appeared in the code: this then required a `\relax` after `\RequirePackage` to prevent an error with “unsafe” definitions as seen for example with `capoptions`. The optional loading here is done using a group and `\ifx` test as we are not quite in the position to have a single name for `\pdfstrcmp` just yet.

```

141 <*package>
142 \begingroup
143   \def\@tempa{LaTeX2e}%
144   \def\next{}%
145   \ifx\fmtname\@tempa
146     \expandafter\ifx\csname extrafloats\endcsname\relax
147       \def\next
148         {%
149           \RequirePackage{etex}%
150           \csname reserveinserts\endcsname{32}%
151         }%
152     \fi
153   \fi
154 \expandafter\endgroup
155 \next
156 </package>

```

1.6 Character data

TeX needs various pieces of data to be set about characters, in particular which ones to treat as letters and which `\lccode` values apply as these affect hyphenation. It makes most sense to set this and related information up in one place. Whilst for LuaTeX hyphenation patterns can be read anywhere, other engines have to build them into the format and so we *must* do this set up before reading the patterns. For the Unicode engines, there are shared loaders available to obtain the relevant information directly from the Unicode Consortium data files. These need standard (Ini)TeX category codes and primitive availability and must therefore be loaded *very* early. This has a knock-on effect on the 8-bit set up: it makes sense to do the definitions for those here as well so it is all in one place.

For X_YTeX and LuaTeX, which are natively Unicode engines, simply load the Unicode data.

```

157 <*initex>
158 \ifdefined\Umathcode
159   \input load-unicode-data %
160   \input load-unicode-math-classes %
161 \else

```

For the 8-bit engines a font encoding scheme must be chosen. At present, this is the EC (T1) scheme, with the assumption that languages for which this is not appropriate will be used with one of the Unicode engines.

```
162 \begingroup
```

Lower case chars: map to themselves when lower casing and down by "20 when upper casing. (The characters a–z are set up correctly by `iniTeX`.)

```
163 \def\temp{%
164   \ifnum\count0>\count2 %
165   \else
166     \global\lccode\count0 = \count0 %
167     \global\uccode\count0 = \numexpr\count0 - "20\relax
168     \advance\count0 by 1 %
169     \expandafter\temp
170   \fi
171 }
172 \count0 = "A0 %
173 \count2 = "BC %
174 \temp
175 \count0 = "E0 %
176 \count2 = "FF %
177 \temp
```

Upper case chars: map up by "20 when lower casing, to themselves when upper casing and require an `\sfcode` of 999. (The characters A–Z are set up correctly by `iniTeX`.)

```
178 \def\temp{%
179   \ifnum\count0>\count2 %
180   \else
181     \global\lccode\count0 = \numexpr\count0 + "20\relax
182     \global\uccode\count0 = \count0 %
183     \global\sffcode\count0 = 999 %
184     \advance\count0 by 1 %
185     \expandafter\temp
186   \fi
187 }
188 \count0 = "80 %
189 \count2 = "9C %
190 \temp
191 \count0 = "C0 %
192 \count2 = "DF %
193 \temp
```

A few special cases where things are not as one might expect using the above pattern: dotless-I, dotless-J, dotted-I and d-bar.

```
194 \global\lccode'\^Y = '\^Y %
195 \global\uccode'\^Y = '\I %
196 \global\lccode'\^Z = '\^Z %
197 \global\uccode'\^Y = '\J %
198 \global\lccode"9D = '\i %
199 \global\uccode"9D = "9D %
200 \global\lccode"9E = "9E %
201 \global\uccode"9E = "D0 %
```

Allow hyphenation at a zero-width glyph (used to break up ligatures or to place accents between characters).

```

202     \global\lccode23 = 23 %
203     \endgroup
204     \fi
205 \</initex>

```

1.7 The L^AT_EX3 code environment

The code environment is now set up.

\ExplSyntaxOff Before changing any category codes, in package mode we need to save the situation before loading. Note the set up here means that once applied `\ExplSyntaxOff` becomes a “do nothing” command until `\ExplSyntaxOn` is used. For format mode, there is no need to save category codes so that step is skipped.

```

206 \protected\def\ExplSyntaxOff{}%
207 \*package)
208 \protected\edef\ExplSyntaxOff
209 {%
210     \protected\def\ExplSyntaxOff{}%
211     \catcode 9 = \the\catcode 9\relax
212     \catcode 32 = \the\catcode 32\relax
213     \catcode 34 = \the\catcode 34\relax
214     \catcode 38 = \the\catcode 38\relax
215     \catcode 58 = \the\catcode 58\relax
216     \catcode 94 = \the\catcode 94\relax
217     \catcode 95 = \the\catcode 95\relax
218     \catcode 124 = \the\catcode 124\relax
219     \catcode 126 = \the\catcode 126\relax
220     \endlinechar = \the\endlinechar\relax
221     \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
222 }%
223 \</package>

```

(End definition for `\ExplSyntaxOff`. This function is documented on page 6.)

The code environment is now set up.

```

224 \catcode 9 = 9\relax
225 \catcode 32 = 9\relax
226 \catcode 34 = 12\relax
227 \catcode 38 = 4\relax
228 \catcode 58 = 11\relax
229 \catcode 94 = 7\relax
230 \catcode 95 = 11\relax
231 \catcode 124 = 12\relax
232 \catcode 126 = 10\relax
233 \endlinechar = 32\relax

```

`\l__kernel_expl_bool` The status for experimental code syntax: this is on at present.

```

234 \chardef\l__kernel_expl_bool = 1\relax

```

(End definition for `\l__kernel_expl_bool`.)

\ExplSyntaxOn The idea here is that multiple `\ExplSyntaxOn` calls are not going to mess up category codes, and that multiple calls to `\ExplSyntaxOff` are also not wasting time. Applying

`\ExplSyntaxOn` alters the definition of `\ExplSyntaxOff` and so in package mode this function should not be used until after the end of the loading process!

```

235 \protected \def \ExplSyntaxOn
236 {
237   \bool_if:NF \l__kernel_expl_bool
238   {
239     \cs_set_protected:Npx \ExplSyntaxOff
240     {
241       \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
242       \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
243       \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
244       \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
245       \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
246       \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
247       \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
248       \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
249       \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
250       \tex_endlinechar:D =
251         \tex_the:D \tex_endlinechar:D \scan_stop:
252       \bool_set_false:N \l__kernel_expl_bool
253       \cs_set_protected:Npn \ExplSyntaxOff { }
254     }
255   }
256   \char_set_catcode_ignore:n { 9 } % tab
257   \char_set_catcode_ignore:n { 32 } % space
258   \char_set_catcode_other:n { 34 } % double quote
259   \char_set_catcode_alignment:n { 38 } % ampersand
260   \char_set_catcode_letter:n { 58 } % colon
261   \char_set_catcode_math_superscript:n { 94 } % circumflex
262   \char_set_catcode_letter:n { 95 } % underscore
263   \char_set_catcode_other:n { 124 } % pipe
264   \char_set_catcode_space:n { 126 } % tilde
265   \tex_endlinechar:D = 32 \scan_stop:
266   \bool_set_true:N \l__kernel_expl_bool
267 }

```

(End definition for `\ExplSyntaxOn`. This function is documented on page 6.)

```

268 \</initex | package>

```

2 l3names implementation

```

269 \<*initex | package>

```

The prefix here is `kernel`. A few places need `@@` to be left as is; this is obtained as `@@@`.

```

270 \@@=kernel

```

The code here simply renames all of the primitives to new, internal, names. In format mode, it also deletes all of the existing names (although some do come back later).

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded csname in the hash table.

```

271 \let \tex_global:D \global

```



```
272 \let \tex_let:D \let
```

Everything is inside a (rather long) group, which keeps `__kernel_primitive:NN` trapped.

```
273 \begingroup
```

`__kernel_primitive:NN` A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```
274 \long \def \__kernel_primitive:NN #1#2
275 {
276   \tex_global:D \tex_let:D #2 #1
277 }*initex>
278 \tex_global:D \tex_let:D #1 \tex_undefined:D
279 </initex>
280 }
```

(End definition for `__kernel_primitive:NN`.)

To allow extracting “just the names”, a bit of DocStrip fiddling.

```
281 </initex | package>
282 }*initex | names | package>
```

In the current incarnation of this package, all TeX primitives are given a new name of the form `\tex_oldname:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```
283 \__kernel_primitive:NN \tex_space:D
284 \__kernel_primitive:NN \tex_italiccorrection:D
285 \__kernel_primitive:NN \tex_hyphen:D
```

Now all the other primitives.

```
286 \__kernel_primitive:NN \tex_above:D
287 \__kernel_primitive:NN \tex_abovedisplayshortskip:D
288 \__kernel_primitive:NN \tex_abovedisplayskip:D
289 \__kernel_primitive:NN \tex_abovewithdelims:D
290 \__kernel_primitive:NN \tex_accent:D
291 \__kernel_primitive:NN \tex_adjdemerits:D
292 \__kernel_primitive:NN \tex_advance:D
293 \__kernel_primitive:NN \tex_afterassignment:D
294 \__kernel_primitive:NN \tex_aftergroup:D
295 \__kernel_primitive:NN \tex_atop:D
296 \__kernel_primitive:NN \tex_atopwithdelims:D
297 \__kernel_primitive:NN \tex_badness:D
298 \__kernel_primitive:NN \tex_baselineskip:D
299 \__kernel_primitive:NN \tex_batchmode:D
300 \__kernel_primitive:NN \tex_begingroup:D
301 \__kernel_primitive:NN \tex_belowdisplayshortskip:D
302 \__kernel_primitive:NN \tex_belowdisplayskip:D
303 \__kernel_primitive:NN \tex_binoppenalty:D
304 \__kernel_primitive:NN \tex_botmark:D
305 \__kernel_primitive:NN \tex_box:D
306 \__kernel_primitive:NN \tex_boxmaxdepth:D
307 \__kernel_primitive:NN \tex_brokenpenalty:D
308 \__kernel_primitive:NN \tex_catcode:D
309 \__kernel_primitive:NN \tex_char:D
310 \__kernel_primitive:NN \tex_chardef:D
```

311	_kernel_primitive:NN	\cleaders	\tex_cleaders:D
312	_kernel_primitive:NN	\closein	\tex_closein:D
313	_kernel_primitive:NN	\closeout	\tex_closeout:D
314	_kernel_primitive:NN	\clubpenalty	\tex_clubpenalty:D
315	_kernel_primitive:NN	\copy	\tex_copy:D
316	_kernel_primitive:NN	\count	\tex_count:D
317	_kernel_primitive:NN	\countdef	\tex_countdef:D
318	_kernel_primitive:NN	\cr	\tex_cr:D
319	_kernel_primitive:NN	\crrcr	\tex_crrcr:D
320	_kernel_primitive:NN	\csname	\tex_csname:D
321	_kernel_primitive:NN	\day	\tex_day:D
322	_kernel_primitive:NN	\deadcycles	\tex_deadcycles:D
323	_kernel_primitive:NN	\def	\tex_def:D
324	_kernel_primitive:NN	\defaultthyphenchar	\tex_defaultthyphenchar:D
325	_kernel_primitive:NN	\defaultskewchar	\tex_defaultskewchar:D
326	_kernel_primitive:NN	\delcode	\tex_delcode:D
327	_kernel_primitive:NN	\delimiter	\tex_delimiter:D
328	_kernel_primitive:NN	\delimiterfactor	\tex_delimiterfactor:D
329	_kernel_primitive:NN	\delimitershortfall	\tex_delimitershortfall:D
330	_kernel_primitive:NN	\dimen	\tex_dimen:D
331	_kernel_primitive:NN	\dimendef	\tex_dimendef:D
332	_kernel_primitive:NN	\discretionary	\tex_discretionary:D
333	_kernel_primitive:NN	\displayindent	\tex_displayindent:D
334	_kernel_primitive:NN	\displaylimits	\tex_displaylimits:D
335	_kernel_primitive:NN	\displaystyle	\tex_displaystyle:D
336	_kernel_primitive:NN	\displaywidowpenalty	\tex_displaywidowpenalty:D
337	_kernel_primitive:NN	\displaywidth	\tex_displaywidth:D
338	_kernel_primitive:NN	\divide	\tex_divide:D
339	_kernel_primitive:NN	\doublehyphendemerits	\tex_doublehyphendemerits:D
340	_kernel_primitive:NN	\dp	\tex_dp:D
341	_kernel_primitive:NN	\dump	\tex_dump:D
342	_kernel_primitive:NN	\edef	\tex_edef:D
343	_kernel_primitive:NN	\else	\tex_else:D
344	_kernel_primitive:NN	\emergencystretch	\tex_emergencystretch:D
345	_kernel_primitive:NN	\end	\tex_end:D
346	_kernel_primitive:NN	\endcsname	\tex_endcsname:D
347	_kernel_primitive:NN	\endgroup	\tex_endgroup:D
348	_kernel_primitive:NN	\endinput	\tex_endinput:D
349	_kernel_primitive:NN	\endlinechar	\tex_endlinechar:D
350	_kernel_primitive:NN	\eqno	\tex_eqno:D
351	_kernel_primitive:NN	\errhelp	\tex_errhelp:D
352	_kernel_primitive:NN	\errmessage	\tex_errmessage:D
353	_kernel_primitive:NN	\errorcontextlines	\tex_errorcontextlines:D
354	_kernel_primitive:NN	\errorstopmode	\tex_errorstopmode:D
355	_kernel_primitive:NN	\escapechar	\tex_escapechar:D
356	_kernel_primitive:NN	\everycr	\tex_everycr:D
357	_kernel_primitive:NN	\everydisplay	\tex_everydisplay:D
358	_kernel_primitive:NN	\everyhbox	\tex_everyhbox:D
359	_kernel_primitive:NN	\everyjob	\tex_everyjob:D
360	_kernel_primitive:NN	\everymath	\tex_everymath:D
361	_kernel_primitive:NN	\everypar	\tex_everypar:D
362	_kernel_primitive:NN	\everyvbox	\tex_everyvbox:D
363	_kernel_primitive:NN	\exhyphenpenalty	\tex_exhyphenpenalty:D
364	_kernel_primitive:NN	\expandafter	\tex_expandafter:D

365	_kernel_primitive:NN \fam	\tex_fam:D
366	_kernel_primitive:NN \fi	\tex_fi:D
367	_kernel_primitive:NN \finalhyphendemerits	\tex_finalhyphendemerits:D
368	_kernel_primitive:NN \firstmark	\tex_firstmark:D
369	_kernel_primitive:NN \floatingpenalty	\tex_floatingpenalty:D
370	_kernel_primitive:NN \font	\tex_font:D
371	_kernel_primitive:NN \fontdimen	\tex_fontdimen:D
372	_kernel_primitive:NN \fontname	\tex_fontname:D
373	_kernel_primitive:NN \futurelet	\tex_futurelet:D
374	_kernel_primitive:NN \gdef	\tex_gdef:D
375	_kernel_primitive:NN \global	\tex_global:D
376	_kernel_primitive:NN \globaldefs	\tex_globaldefs:D
377	_kernel_primitive:NN \halign	\tex_halign:D
378	_kernel_primitive:NN \hangafter	\tex_hangafter:D
379	_kernel_primitive:NN \hangindent	\tex_hangindent:D
380	_kernel_primitive:NN \hbadness	\tex_hbadness:D
381	_kernel_primitive:NN \hbox	\tex_hbox:D
382	_kernel_primitive:NN \hfil	\tex_hfil:D
383	_kernel_primitive:NN \hfill	\tex_hfill:D
384	_kernel_primitive:NN \hfilneg	\tex_hfilneg:D
385	_kernel_primitive:NN \hfuzz	\tex_hfuzz:D
386	_kernel_primitive:NN \hoffset	\tex_hoffset:D
387	_kernel_primitive:NN \holdinginserts	\tex_holdinginserts:D
388	_kernel_primitive:NN \hrule	\tex_hrule:D
389	_kernel_primitive:NN \hsize	\tex_hsize:D
390	_kernel_primitive:NN \hskip	\tex_hskip:D
391	_kernel_primitive:NN \hss	\tex_hss:D
392	_kernel_primitive:NN \ht	\tex_ht:D
393	_kernel_primitive:NN \hyphenation	\tex_hyphenation:D
394	_kernel_primitive:NN \hyphenchar	\tex_hyphenchar:D
395	_kernel_primitive:NN \hyphenpenalty	\tex_hyphenpenalty:D
396	_kernel_primitive:NN \if	\tex_if:D
397	_kernel_primitive:NN \ifcase	\tex_ifcase:D
398	_kernel_primitive:NN \ifcat	\tex_ifcat:D
399	_kernel_primitive:NN \ifdim	\tex_ifdim:D
400	_kernel_primitive:NN \ifeof	\tex_ifeof:D
401	_kernel_primitive:NN \iffalse	\tex_iffalse:D
402	_kernel_primitive:NN \ifhbox	\tex_ifhbox:D
403	_kernel_primitive:NN \ifhmode	\tex_ifhmode:D
404	_kernel_primitive:NN \ifinner	\tex_ifinner:D
405	_kernel_primitive:NN \ifmmode	\tex_ifmmode:D
406	_kernel_primitive:NN \ifnum	\tex_ifnum:D
407	_kernel_primitive:NN \ifodd	\tex_ifodd:D
408	_kernel_primitive:NN \iftrue	\tex_iftrue:D
409	_kernel_primitive:NN \ifvbox	\tex_ifvbox:D
410	_kernel_primitive:NN \ifvmode	\tex_ifvmode:D
411	_kernel_primitive:NN \ifvoid	\tex_ifvoid:D
412	_kernel_primitive:NN \ifx	\tex_ifx:D
413	_kernel_primitive:NN \ignorespaces	\tex_ignorespaces:D
414	_kernel_primitive:NN \immediate	\tex_immediate:D
415	_kernel_primitive:NN \indent	\tex_indent:D
416	_kernel_primitive:NN \input	\tex_input:D
417	_kernel_primitive:NN \inputlineno	\tex_inputlineno:D
418	_kernel_primitive:NN \insert	\tex_insert:D

419	_kernel_primitive:NN	\insertpenalties	\tex_insertpenalties:D
420	_kernel_primitive:NN	\interlinepenalty	\tex_interlinepenalty:D
421	_kernel_primitive:NN	\jobname	\tex_jobname:D
422	_kernel_primitive:NN	\kern	\tex_kern:D
423	_kernel_primitive:NN	\language	\tex_language:D
424	_kernel_primitive:NN	\lastbox	\tex_lastbox:D
425	_kernel_primitive:NN	\lastkern	\tex_lastkern:D
426	_kernel_primitive:NN	\lastpenalty	\tex_lastpenalty:D
427	_kernel_primitive:NN	\lastskip	\tex_lastskip:D
428	_kernel_primitive:NN	\lccode	\tex_lccode:D
429	_kernel_primitive:NN	\leaders	\tex_leaders:D
430	_kernel_primitive:NN	\left	\tex_left:D
431	_kernel_primitive:NN	\lefthyphenmin	\tex_lefthyphenmin:D
432	_kernel_primitive:NN	\leftskip	\tex_leftskip:D
433	_kernel_primitive:NN	\leqno	\tex_leqno:D
434	_kernel_primitive:NN	\let	\tex_let:D
435	_kernel_primitive:NN	\limits	\tex_limits:D
436	_kernel_primitive:NN	\linepenalty	\tex_linepenalty:D
437	_kernel_primitive:NN	\lineskip	\tex_lineskip:D
438	_kernel_primitive:NN	\lineskiplimit	\tex_lineskiplimit:D
439	_kernel_primitive:NN	\long	\tex_long:D
440	_kernel_primitive:NN	\looseness	\tex_looseness:D
441	_kernel_primitive:NN	\lower	\tex_lower:D
442	_kernel_primitive:NN	\lowercase	\tex_lowercase:D
443	_kernel_primitive:NN	\mag	\tex_mag:D
444	_kernel_primitive:NN	\mark	\tex_mark:D
445	_kernel_primitive:NN	\mathaccent	\tex_mathaccent:D
446	_kernel_primitive:NN	\mathbin	\tex_mathbin:D
447	_kernel_primitive:NN	\mathchar	\tex_mathchar:D
448	_kernel_primitive:NN	\mathchardef	\tex_mathchardef:D
449	_kernel_primitive:NN	\mathchoice	\tex_mathchoice:D
450	_kernel_primitive:NN	\mathclose	\tex_mathclose:D
451	_kernel_primitive:NN	\mathcode	\tex_mathcode:D
452	_kernel_primitive:NN	\mathinner	\tex_mathinner:D
453	_kernel_primitive:NN	\mathop	\tex_mathop:D
454	_kernel_primitive:NN	\mathopen	\tex_mathopen:D
455	_kernel_primitive:NN	\mathord	\tex_mathord:D
456	_kernel_primitive:NN	\mathpunct	\tex_mathpunct:D
457	_kernel_primitive:NN	\mathrel	\tex_mathrel:D
458	_kernel_primitive:NN	\mathsurround	\tex_mathsurround:D
459	_kernel_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
460	_kernel_primitive:NN	\maxdepth	\tex_maxdepth:D
461	_kernel_primitive:NN	\meaning	\tex_meaning:D
462	_kernel_primitive:NN	\medmuskip	\tex_medmuskip:D
463	_kernel_primitive:NN	\message	\tex_message:D
464	_kernel_primitive:NN	\mkern	\tex_mkern:D
465	_kernel_primitive:NN	\month	\tex_month:D
466	_kernel_primitive:NN	\moveleft	\tex_moveleft:D
467	_kernel_primitive:NN	\moveright	\tex_moveright:D
468	_kernel_primitive:NN	\mskip	\tex_mskip:D
469	_kernel_primitive:NN	\multiply	\tex_multiply:D
470	_kernel_primitive:NN	\muskip	\tex_muskip:D
471	_kernel_primitive:NN	\muskipdef	\tex_muskipdef:D
472	_kernel_primitive:NN	\newlinechar	\tex_newlinechar:D

473	<code>_kernel_primitive:NN</code>	<code>\noalign</code>	<code>\tex_noalign:D</code>
474	<code>_kernel_primitive:NN</code>	<code>\noboundary</code>	<code>\tex_noboundary:D</code>
475	<code>_kernel_primitive:NN</code>	<code>\noexpand</code>	<code>\tex_noexpand:D</code>
476	<code>_kernel_primitive:NN</code>	<code>\noindent</code>	<code>\tex_noindent:D</code>
477	<code>_kernel_primitive:NN</code>	<code>\nolimits</code>	<code>\tex_nolimits:D</code>
478	<code>_kernel_primitive:NN</code>	<code>\nonscript</code>	<code>\tex_nonscript:D</code>
479	<code>_kernel_primitive:NN</code>	<code>\nonstopmode</code>	<code>\tex_nonstopmode:D</code>
480	<code>_kernel_primitive:NN</code>	<code>\nulldelimiterspace</code>	<code>\tex_nulldelimiterspace:D</code>
481	<code>_kernel_primitive:NN</code>	<code>\nullfont</code>	<code>\tex_nullfont:D</code>
482	<code>_kernel_primitive:NN</code>	<code>\number</code>	<code>\tex_number:D</code>
483	<code>_kernel_primitive:NN</code>	<code>\omit</code>	<code>\tex_omit:D</code>
484	<code>_kernel_primitive:NN</code>	<code>\openin</code>	<code>\tex_openin:D</code>
485	<code>_kernel_primitive:NN</code>	<code>\openout</code>	<code>\tex_openout:D</code>
486	<code>_kernel_primitive:NN</code>	<code>\or</code>	<code>\tex_or:D</code>
487	<code>_kernel_primitive:NN</code>	<code>\outer</code>	<code>\tex_outer:D</code>
488	<code>_kernel_primitive:NN</code>	<code>\output</code>	<code>\tex_output:D</code>
489	<code>_kernel_primitive:NN</code>	<code>\outputpenalty</code>	<code>\tex_outputpenalty:D</code>
490	<code>_kernel_primitive:NN</code>	<code>\over</code>	<code>\tex_over:D</code>
491	<code>_kernel_primitive:NN</code>	<code>\overfullrule</code>	<code>\tex_overfullrule:D</code>
492	<code>_kernel_primitive:NN</code>	<code>\overline</code>	<code>\tex_overline:D</code>
493	<code>_kernel_primitive:NN</code>	<code>\overwithdelims</code>	<code>\tex_overwithdelims:D</code>
494	<code>_kernel_primitive:NN</code>	<code>\pagedepth</code>	<code>\tex_pagedepth:D</code>
495	<code>_kernel_primitive:NN</code>	<code>\pagefilllstretch</code>	<code>\tex_pagefilllstretch:D</code>
496	<code>_kernel_primitive:NN</code>	<code>\pagefillstretch</code>	<code>\tex_pagefillstretch:D</code>
497	<code>_kernel_primitive:NN</code>	<code>\pagefilstretch</code>	<code>\tex_pagefilstretch:D</code>
498	<code>_kernel_primitive:NN</code>	<code>\pagegoal</code>	<code>\tex_pagegoal:D</code>
499	<code>_kernel_primitive:NN</code>	<code>\pageshrink</code>	<code>\tex_pageshrink:D</code>
500	<code>_kernel_primitive:NN</code>	<code>\pagestretch</code>	<code>\tex_pagestretch:D</code>
501	<code>_kernel_primitive:NN</code>	<code>\pagetotal</code>	<code>\tex_pagetotal:D</code>
502	<code>_kernel_primitive:NN</code>	<code>\par</code>	<code>\tex_par:D</code>
503	<code>_kernel_primitive:NN</code>	<code>\parfillskip</code>	<code>\tex_parfillskip:D</code>
504	<code>_kernel_primitive:NN</code>	<code>\parindent</code>	<code>\tex_parindent:D</code>
505	<code>_kernel_primitive:NN</code>	<code>\parshape</code>	<code>\tex_parshape:D</code>
506	<code>_kernel_primitive:NN</code>	<code>\parskip</code>	<code>\tex_parskip:D</code>
507	<code>_kernel_primitive:NN</code>	<code>\patterns</code>	<code>\tex_patterns:D</code>
508	<code>_kernel_primitive:NN</code>	<code>\pausing</code>	<code>\tex_pausing:D</code>
509	<code>_kernel_primitive:NN</code>	<code>\penalty</code>	<code>\tex_penalty:D</code>
510	<code>_kernel_primitive:NN</code>	<code>\postdisplaypenalty</code>	<code>\tex_postdisplaypenalty:D</code>
511	<code>_kernel_primitive:NN</code>	<code>\predisplaypenalty</code>	<code>\tex_predisplaypenalty:D</code>
512	<code>_kernel_primitive:NN</code>	<code>\predplaysize</code>	<code>\tex_predplaysize:D</code>
513	<code>_kernel_primitive:NN</code>	<code>\pretolerance</code>	<code>\tex_pretolerance:D</code>
514	<code>_kernel_primitive:NN</code>	<code>\prevdepth</code>	<code>\tex_prevdepth:D</code>
515	<code>_kernel_primitive:NN</code>	<code>\prevgraf</code>	<code>\tex_prevgraf:D</code>
516	<code>_kernel_primitive:NN</code>	<code>\radical</code>	<code>\tex_radical:D</code>
517	<code>_kernel_primitive:NN</code>	<code>\raise</code>	<code>\tex_raise:D</code>
518	<code>_kernel_primitive:NN</code>	<code>\read</code>	<code>\tex_read:D</code>
519	<code>_kernel_primitive:NN</code>	<code>\relax</code>	<code>\tex_relax:D</code>
520	<code>_kernel_primitive:NN</code>	<code>\relpenalty</code>	<code>\tex_relpenalty:D</code>
521	<code>_kernel_primitive:NN</code>	<code>\right</code>	<code>\tex_right:D</code>
522	<code>_kernel_primitive:NN</code>	<code>\righthyphenmin</code>	<code>\tex_righthyphenmin:D</code>
523	<code>_kernel_primitive:NN</code>	<code>\rightskip</code>	<code>\tex_rightskip:D</code>
524	<code>_kernel_primitive:NN</code>	<code>\romannumeral</code>	<code>\tex_romannumeral:D</code>
525	<code>_kernel_primitive:NN</code>	<code>\scriptfont</code>	<code>\tex_scriptfont:D</code>
526	<code>_kernel_primitive:NN</code>	<code>\scriptscriptfont</code>	<code>\tex_scriptscriptfont:D</code>

527	<code>__kernel_primitive:NN \scriptscriptstyle</code>	<code>\tex_scriptscriptstyle:D</code>
528	<code>__kernel_primitive:NN \scriptspace</code>	<code>\tex_scriptspace:D</code>
529	<code>__kernel_primitive:NN \scriptstyle</code>	<code>\tex_scriptstyle:D</code>
530	<code>__kernel_primitive:NN \scrollmode</code>	<code>\tex_scrollmode:D</code>
531	<code>__kernel_primitive:NN \setbox</code>	<code>\tex_setbox:D</code>
532	<code>__kernel_primitive:NN \setlanguage</code>	<code>\tex_setlanguage:D</code>
533	<code>__kernel_primitive:NN \sfcode</code>	<code>\tex_sfcode:D</code>
534	<code>__kernel_primitive:NN \shipout</code>	<code>\tex_shipout:D</code>
535	<code>__kernel_primitive:NN \show</code>	<code>\tex_show:D</code>
536	<code>__kernel_primitive:NN \showbox</code>	<code>\tex_showbox:D</code>
537	<code>__kernel_primitive:NN \showboxbreadth</code>	<code>\tex_showboxbreadth:D</code>
538	<code>__kernel_primitive:NN \showboxdepth</code>	<code>\tex_showboxdepth:D</code>
539	<code>__kernel_primitive:NN \showlists</code>	<code>\tex_showlists:D</code>
540	<code>__kernel_primitive:NN \showthe</code>	<code>\tex_showthe:D</code>
541	<code>__kernel_primitive:NN \skewchar</code>	<code>\tex_skewchar:D</code>
542	<code>__kernel_primitive:NN \skip</code>	<code>\tex_skip:D</code>
543	<code>__kernel_primitive:NN \skipdef</code>	<code>\tex_skipdef:D</code>
544	<code>__kernel_primitive:NN \spacefactor</code>	<code>\tex_spacefactor:D</code>
545	<code>__kernel_primitive:NN \spaceskip</code>	<code>\tex_spaceskip:D</code>
546	<code>__kernel_primitive:NN \span</code>	<code>\tex_span:D</code>
547	<code>__kernel_primitive:NN \special</code>	<code>\tex_special:D</code>
548	<code>__kernel_primitive:NN \splitbotmark</code>	<code>\tex_splitbotmark:D</code>
549	<code>__kernel_primitive:NN \splitfirstmark</code>	<code>\tex_splitfirstmark:D</code>
550	<code>__kernel_primitive:NN \splitmaxdepth</code>	<code>\tex_splitmaxdepth:D</code>
551	<code>__kernel_primitive:NN \splittopskip</code>	<code>\tex_splittopskip:D</code>
552	<code>__kernel_primitive:NN \string</code>	<code>\tex_string:D</code>
553	<code>__kernel_primitive:NN \tabskip</code>	<code>\tex_tabskip:D</code>
554	<code>__kernel_primitive:NN \textfont</code>	<code>\tex_textfont:D</code>
555	<code>__kernel_primitive:NN \textstyle</code>	<code>\tex_textstyle:D</code>
556	<code>__kernel_primitive:NN \the</code>	<code>\tex_the:D</code>
557	<code>__kernel_primitive:NN \thickmuskip</code>	<code>\tex_thickmuskip:D</code>
558	<code>__kernel_primitive:NN \thinmuskip</code>	<code>\tex_thinmuskip:D</code>
559	<code>__kernel_primitive:NN \time</code>	<code>\tex_time:D</code>
560	<code>__kernel_primitive:NN \toks</code>	<code>\tex_toks:D</code>
561	<code>__kernel_primitive:NN \toksdef</code>	<code>\tex_toksdef:D</code>
562	<code>__kernel_primitive:NN \tolerance</code>	<code>\tex_tolerance:D</code>
563	<code>__kernel_primitive:NN \topmark</code>	<code>\tex_topmark:D</code>
564	<code>__kernel_primitive:NN \topskip</code>	<code>\tex_topskip:D</code>
565	<code>__kernel_primitive:NN \tracingcommands</code>	<code>\tex_tracingcommands:D</code>
566	<code>__kernel_primitive:NN \tracinglostchars</code>	<code>\tex_tracinglostchars:D</code>
567	<code>__kernel_primitive:NN \tracingmacros</code>	<code>\tex_tracingmacros:D</code>
568	<code>__kernel_primitive:NN \tracingonline</code>	<code>\tex_tracingonline:D</code>
569	<code>__kernel_primitive:NN \tracingoutput</code>	<code>\tex_tracingoutput:D</code>
570	<code>__kernel_primitive:NN \tracingpages</code>	<code>\tex_tracingpages:D</code>
571	<code>__kernel_primitive:NN \tracingparagraphs</code>	<code>\tex_tracingparagraphs:D</code>
572	<code>__kernel_primitive:NN \tracingrestores</code>	<code>\tex_tracingrestores:D</code>
573	<code>__kernel_primitive:NN \tracingstats</code>	<code>\tex_tracingstats:D</code>
574	<code>__kernel_primitive:NN \uccode</code>	<code>\tex_uccode:D</code>
575	<code>__kernel_primitive:NN \uchyph</code>	<code>\tex_uchyph:D</code>
576	<code>__kernel_primitive:NN \underline</code>	<code>\tex_underline:D</code>
577	<code>__kernel_primitive:NN \unhbox</code>	<code>\tex_unhbox:D</code>
578	<code>__kernel_primitive:NN \unhcopy</code>	<code>\tex_unhcopy:D</code>
579	<code>__kernel_primitive:NN \unkern</code>	<code>\tex_unkern:D</code>
580	<code>__kernel_primitive:NN \unpenalty</code>	<code>\tex_unpenalty:D</code>

581	<code>__kernel_primitive:NN \unskip</code>	<code>\tex_unskip:D</code>
582	<code>__kernel_primitive:NN \unvbox</code>	<code>\tex_unvbox:D</code>
583	<code>__kernel_primitive:NN \unvcopy</code>	<code>\tex_unvcopy:D</code>
584	<code>__kernel_primitive:NN \uppercase</code>	<code>\tex_uppercase:D</code>
585	<code>__kernel_primitive:NN \vadjust</code>	<code>\tex_vadjust:D</code>
586	<code>__kernel_primitive:NN \valign</code>	<code>\tex_valign:D</code>
587	<code>__kernel_primitive:NN \vbadness</code>	<code>\tex_vbadness:D</code>
588	<code>__kernel_primitive:NN \vbox</code>	<code>\tex_vbox:D</code>
589	<code>__kernel_primitive:NN \vcenter</code>	<code>\tex_vcenter:D</code>
590	<code>__kernel_primitive:NN \vfil</code>	<code>\tex_vfil:D</code>
591	<code>__kernel_primitive:NN \vfill</code>	<code>\tex_vfill:D</code>
592	<code>__kernel_primitive:NN \vfилneg</code>	<code>\tex_vfilneg:D</code>
593	<code>__kernel_primitive:NN \vfuzz</code>	<code>\tex_vfuzz:D</code>
594	<code>__kernel_primitive:NN \voffset</code>	<code>\tex_voffset:D</code>
595	<code>__kernel_primitive:NN \vrule</code>	<code>\tex_vrule:D</code>
596	<code>__kernel_primitive:NN \vsize</code>	<code>\tex_vsize:D</code>
597	<code>__kernel_primitive:NN \vskip</code>	<code>\tex_vskip:D</code>
598	<code>__kernel_primitive:NN \vsplit</code>	<code>\tex_vsplit:D</code>
599	<code>__kernel_primitive:NN \vss</code>	<code>\tex_vss:D</code>
600	<code>__kernel_primitive:NN \vtop</code>	<code>\tex_vtop:D</code>
601	<code>__kernel_primitive:NN \wd</code>	<code>\tex_wd:D</code>
602	<code>__kernel_primitive:NN \widowpenalty</code>	<code>\tex_widowpenalty:D</code>
603	<code>__kernel_primitive:NN \write</code>	<code>\tex_write:D</code>
604	<code>__kernel_primitive:NN \xdef</code>	<code>\tex_xdef:D</code>
605	<code>__kernel_primitive:NN \xleaders</code>	<code>\tex_xleaders:D</code>
606	<code>__kernel_primitive:NN \xspaceskip</code>	<code>\tex_xspaceskip:D</code>
607	<code>__kernel_primitive:NN \year</code>	<code>\tex_year:D</code>

Since L^AT_EX 3 requires at least the ε -T_EX extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

608	<code>__kernel_primitive:NN \beginL</code>	<code>\etex_beginL:D</code>
609	<code>__kernel_primitive:NN \beginR</code>	<code>\etex_beginR:D</code>
610	<code>__kernel_primitive:NN \botmarks</code>	<code>\etex_botmarks:D</code>
611	<code>__kernel_primitive:NN \clubpenalties</code>	<code>\etex_clubpenalties:D</code>
612	<code>__kernel_primitive:NN \currentgrouplevel</code>	<code>\etex_currentgrouplevel:D</code>
613	<code>__kernel_primitive:NN \currentgrouptype</code>	<code>\etex_currentgrouptype:D</code>
614	<code>__kernel_primitive:NN \currentifbranch</code>	<code>\etex_currentifbranch:D</code>
615	<code>__kernel_primitive:NN \currentiflevel</code>	<code>\etex_currentiflevel:D</code>
616	<code>__kernel_primitive:NN \currentifttype</code>	<code>\etex_currentifttype:D</code>
617	<code>__kernel_primitive:NN \detokenize</code>	<code>\etex_detokenize:D</code>
618	<code>__kernel_primitive:NN \dimexpr</code>	<code>\etex_dimexpr:D</code>
619	<code>__kernel_primitive:NN \displaywidowpenalties</code>	
620	<code>\etex_displaywidowpenalties:D</code>	
621	<code>__kernel_primitive:NN \endL</code>	<code>\etex_endL:D</code>
622	<code>__kernel_primitive:NN \endR</code>	<code>\etex_endR:D</code>
623	<code>__kernel_primitive:NN \eTeXrevision</code>	<code>\etex_eTeXrevision:D</code>
624	<code>__kernel_primitive:NN \eTeXversion</code>	<code>\etex_eTeXversion:D</code>
625	<code>__kernel_primitive:NN \everyeof</code>	<code>\etex_everyeof:D</code>
626	<code>__kernel_primitive:NN \firstmarks</code>	<code>\etex_firstmarks:D</code>
627	<code>__kernel_primitive:NN \fontcharpd</code>	<code>\etex_fontcharpd:D</code>
628	<code>__kernel_primitive:NN \fontcharht</code>	<code>\etex_fontcharht:D</code>
629	<code>__kernel_primitive:NN \fontcharic</code>	<code>\etex_fontcharic:D</code>
630	<code>__kernel_primitive:NN \fontcharwd</code>	<code>\etex_fontcharwd:D</code>
631	<code>__kernel_primitive:NN \glueexpr</code>	<code>\etex_glueexpr:D</code>

632	<code>__kernel_primitive:NN \glueshrink</code>	<code>\etex_glueshrink:D</code>
633	<code>__kernel_primitive:NN \glueshrinkorder</code>	<code>\etex_glueshrinkorder:D</code>
634	<code>__kernel_primitive:NN \gluestretch</code>	<code>\etex_gluestretch:D</code>
635	<code>__kernel_primitive:NN \gluestretchorder</code>	<code>\etex_gluestretchorder:D</code>
636	<code>__kernel_primitive:NN \gluetomu</code>	<code>\etex_gluetomu:D</code>
637	<code>__kernel_primitive:NN \ifcsname</code>	<code>\etex_ifcsname:D</code>
638	<code>__kernel_primitive:NN \ifdefined</code>	<code>\etex_ifdefined:D</code>
639	<code>__kernel_primitive:NN \iffontchar</code>	<code>\etex_iffontchar:D</code>
640	<code>__kernel_primitive:NN \interactionmode</code>	<code>\etex_interactionmode:D</code>
641	<code>__kernel_primitive:NN \interlinepenalties</code>	<code>\etex_interlinepenalties:D</code>
642	<code>__kernel_primitive:NN \lastlinefit</code>	<code>\etex_lastlinefit:D</code>
643	<code>__kernel_primitive:NN \lastnodetype</code>	<code>\etex_lastnodetype:D</code>
644	<code>__kernel_primitive:NN \marks</code>	<code>\etex_marks:D</code>
645	<code>__kernel_primitive:NN \middle</code>	<code>\etex_middle:D</code>
646	<code>__kernel_primitive:NN \muexpr</code>	<code>\etex_muexpr:D</code>
647	<code>__kernel_primitive:NN \mutoglu</code>	<code>\etex_mutoglu:D</code>
648	<code>__kernel_primitive:NN \numexpr</code>	<code>\etex_numexpr:D</code>
649	<code>__kernel_primitive:NN \pagediscards</code>	<code>\etex_pagediscards:D</code>
650	<code>__kernel_primitive:NN \parshapedimen</code>	<code>\etex_parshapedimen:D</code>
651	<code>__kernel_primitive:NN \parshapeindent</code>	<code>\etex_parshapeindent:D</code>
652	<code>__kernel_primitive:NN \parshapelength</code>	<code>\etex_parshapelength:D</code>
653	<code>__kernel_primitive:NN \predisplaydirection</code>	<code>\etex_predisplaydirection:D</code>
654	<code>__kernel_primitive:NN \protected</code>	<code>\etex_protected:D</code>
655	<code>__kernel_primitive:NN \readline</code>	<code>\etex_readline:D</code>
656	<code>__kernel_primitive:NN \savinghyphcodes</code>	<code>\etex_savinghyphcodes:D</code>
657	<code>__kernel_primitive:NN \savingvdiscards</code>	<code>\etex_savingvdiscards:D</code>
658	<code>__kernel_primitive:NN \scantokens</code>	<code>\etex_scantokens:D</code>
659	<code>__kernel_primitive:NN \showgroups</code>	<code>\etex_showgroups:D</code>
660	<code>__kernel_primitive:NN \showifs</code>	<code>\etex_showifs:D</code>
661	<code>__kernel_primitive:NN \showtokens</code>	<code>\etex_showtokens:D</code>
662	<code>__kernel_primitive:NN \splitbotmarks</code>	<code>\etex_splitbotmarks:D</code>
663	<code>__kernel_primitive:NN \splitdiscards</code>	<code>\etex_splitdiscards:D</code>
664	<code>__kernel_primitive:NN \splitfirstmarks</code>	<code>\etex_splitfirstmarks:D</code>
665	<code>__kernel_primitive:NN \TeXXeTstate</code>	<code>\etex_TeXeTstate:D</code>
666	<code>__kernel_primitive:NN \topmarks</code>	<code>\etex_topmarks:D</code>
667	<code>__kernel_primitive:NN \tracingassigns</code>	<code>\etex_tracingassigns:D</code>
668	<code>__kernel_primitive:NN \tracinggroups</code>	<code>\etex_tracinggroups:D</code>
669	<code>__kernel_primitive:NN \tracingifs</code>	<code>\etex_tracingifs:D</code>
670	<code>__kernel_primitive:NN \tracingnesting</code>	<code>\etex_tracingnesting:D</code>
671	<code>__kernel_primitive:NN \tracingscantokens</code>	<code>\etex_tracingscantokens:D</code>
672	<code>__kernel_primitive:NN \unexpanded</code>	<code>\etex_unexpanded:D</code>
673	<code>__kernel_primitive:NN \unless</code>	<code>\etex_unless:D</code>
674	<code>__kernel_primitive:NN \widowpenalties</code>	<code>\etex_widowpenalties:D</code>

The newer primitives are more complex: there are an awful lot of them, and we don't use them all at the moment. So the following is selective, based on those also available in LuaTeX or used in expl3. In the case of the pdfTeX primitives, we retain `pdf` at the start of the names *only* for directly PDF-related primitives, as there are a lot of pdfTeX primitives that start `\pdf...` but are not related to PDF output. These ones related to PDF output or only work in PDF mode.

675	<code>__kernel_primitive:NN \pdfannot</code>	<code>\pdftex_pdfannot:D</code>
676	<code>__kernel_primitive:NN \pdfcatalog</code>	<code>\pdftex_pdfcatalog:D</code>
677	<code>__kernel_primitive:NN \pdfcompresslevel</code>	<code>\pdftex_pdfcompresslevel:D</code>
678	<code>__kernel_primitive:NN \pdfcolorstack</code>	<code>\pdftex_pdfcolorstack:D</code>

679	_kernel_primitive:NN	\pdfcolorstackinit	\pdfTEX_pdfcolorstackinit:D
680	_kernel_primitive:NN	\pdfcreationdate	\pdfTEX_pdfcreationdate:D
681	_kernel_primitive:NN	\pdfdecimaldigits	\pdfTEX_pdfdecimaldigits:D
682	_kernel_primitive:NN	\pdfdest	\pdfTEX_pdfdest:D
683	_kernel_primitive:NN	\pdfdestmargin	\pdfTEX_pdfdestmargin:D
684	_kernel_primitive:NN	\pdfendlink	\pdfTEX_pdfendlink:D
685	_kernel_primitive:NN	\pdfendthread	\pdfTEX_pdfendthread:D
686	_kernel_primitive:NN	\pdffontattr	\pdfTEX_pdffontattr:D
687	_kernel_primitive:NN	\pdffontname	\pdfTEX_pdffontname:D
688	_kernel_primitive:NN	\pdffontobjnum	\pdfTEX_pdffontobjnum:D
689	_kernel_primitive:NN	\pdfgamma	\pdfTEX_pdfgamma:D
690	_kernel_primitive:NN	\pdfimageapplygamma	\pdfTEX_pdfimageapplygamma:D
691	_kernel_primitive:NN	\pdfimagegamma	\pdfTEX_pdfimagegamma:D
692	_kernel_primitive:NN	\pdfgentounicode	\pdfTEX_pdfgentounicode:D
693	_kernel_primitive:NN	\pdfglyptounicode	\pdfTEX_pdfglyptounicode:D
694	_kernel_primitive:NN	\pdfhorigin	\pdfTEX_pdfhorigin:D
695	_kernel_primitive:NN	\pdfimagehicolor	\pdfTEX_pdfimagehicolor:D
696	_kernel_primitive:NN	\pdfimageresolution	\pdfTEX_pdfimageresolution:D
697	_kernel_primitive:NN	\pdfincludechars	\pdfTEX_pdfincludechars:D
698	_kernel_primitive:NN	\pdfinclusioncopyfonts	
699		\pdfTEX_pdfinclusioncopyfonts:D	
700	_kernel_primitive:NN	\pdfinclusionerrorlevel	
701		\pdfTEX_pdfinclusionerrorlevel:D	
702	_kernel_primitive:NN	\pdfinfo	\pdfTEX_pdfinfo:D
703	_kernel_primitive:NN	\pdflastannot	\pdfTEX_pdflastannot:D
704	_kernel_primitive:NN	\pdflastlink	\pdfTEX_pdflastlink:D
705	_kernel_primitive:NN	\pdflastobj	\pdfTEX_pdflastobj:D
706	_kernel_primitive:NN	\pdflastxform	\pdfTEX_pdflastxform:D
707	_kernel_primitive:NN	\pdflastximage	\pdfTEX_pdflastximage:D
708	_kernel_primitive:NN	\pdflastximagecolordepth	
709		\pdfTEX_pdflastximagecolordepth:D	
710	_kernel_primitive:NN	\pdflastximagepages	\pdfTEX_pdflastximagepages:D
711	_kernel_primitive:NN	\pdflinkmargin	\pdfTEX_pdflinkmargin:D
712	_kernel_primitive:NN	\pdfliteral	\pdfTEX_pdfliteral:D
713	_kernel_primitive:NN	\pdfminorversion	\pdfTEX_pdfminorversion:D
714	_kernel_primitive:NN	\pdfnames	\pdfTEX_pdfnames:D
715	_kernel_primitive:NN	\pdfobj	\pdfTEX_pdfobj:D
716	_kernel_primitive:NN	\pdfobjcompresslevel	
717		\pdfTEX_pdfobjcompresslevel:D	
718	_kernel_primitive:NN	\pdfoutline	\pdfTEX_pdfoutline:D
719	_kernel_primitive:NN	\pdfoutput	\pdfTEX_pdfoutput:D
720	_kernel_primitive:NN	\pdfpageattr	\pdfTEX_pdfpageattr:D
721	_kernel_primitive:NN	\pdfpagebox	\pdfTEX_pdfpagebox:D
722	_kernel_primitive:NN	\pdfpageref	\pdfTEX_pdfpageref:D
723	_kernel_primitive:NN	\pdfpageresources	\pdfTEX_pdfpageresources:D
724	_kernel_primitive:NN	\pdfpagesattr	\pdfTEX_pdfpagesattr:D
725	_kernel_primitive:NN	\pdfrefobj	\pdfTEX_pdfrefobj:D
726	_kernel_primitive:NN	\pdfrefxform	\pdfTEX_pdfrefxform:D
727	_kernel_primitive:NN	\pdfrefximage	\pdfTEX_pdfrefximage:D
728	_kernel_primitive:NN	\pdfrestore	\pdfTEX_pdfrestore:D
729	_kernel_primitive:NN	\pdfretval	\pdfTEX_pdfretval:D
730	_kernel_primitive:NN	\pdfsave	\pdfTEX_pdfsave:D
731	_kernel_primitive:NN	\pdfsetmatrix	\pdfTEX_pdfsetmatrix:D
732	_kernel_primitive:NN	\pdfstartlink	\pdfTEX_pdfstartlink:D

733	_kernel_primitive:NN	\pdfstartthread	\pdfTEX_pdfstartthread:D
734	_kernel_primitive:NN	\pdfsuppressptexinfo	
735		\pdfTEX_pdfsuppressptexinfo:D	
736	_kernel_primitive:NN	\pdfthread	\pdfTEX_pdfthread:D
737	_kernel_primitive:NN	\pdfthreadmargin	\pdfTEX_pdfthreadmargin:D
738	_kernel_primitive:NN	\pdftrailer	\pdfTEX_pdftrailer:D
739	_kernel_primitive:NN	\pdfuniqueresname	\pdfTEX_pdfuniqueresname:D
740	_kernel_primitive:NN	\pdfvorigin	\pdfTEX_pdfvorigin:D
741	_kernel_primitive:NN	\pdfxform	\pdfTEX_pdfxform:D
742	_kernel_primitive:NN	\pdfxformattr	\pdfTEX_pdfxformattr:D
743	_kernel_primitive:NN	\pdfxformname	\pdfTEX_pdfxformname:D
744	_kernel_primitive:NN	\pdfxformresources	\pdfTEX_pdfxformresources:D
745	_kernel_primitive:NN	\pdfximage	\pdfTEX_pdfximage:D
746	_kernel_primitive:NN	\pdfximagebbox	\pdfTEX_pdfximagebbox:D

While these are not.

747	_kernel_primitive:NN	\ifpdfabsdim	\pdfTEX_ifabsdim:D
748	_kernel_primitive:NN	\ifpdfabsnum	\pdfTEX_ifabsnum:D
749	_kernel_primitive:NN	\ifpdfprimitive	\pdfTEX_ifprimitive:D
750	_kernel_primitive:NN	\pdfadjustspacing	\pdfTEX_adjustspacing:D
751	_kernel_primitive:NN	\pdfcopyfont	\pdfTEX_copyfont:D
752	_kernel_primitive:NN	\pdfdraftmode	\pdfTEX_draftmode:D
753	_kernel_primitive:NN	\pdfeachlinedepth	\pdfTEX_eachlinedepth:D
754	_kernel_primitive:NN	\pdfeachlineheight	\pdfTEX_eachlineheight:D
755	_kernel_primitive:NN	\pdffilemoddate	\pdfTEX_filemoddate:D
756	_kernel_primitive:NN	\pdffilesize	\pdfTEX_filesize:D
757	_kernel_primitive:NN	\pdffirstlineheight	\pdfTEX_firstlineheight:D
758	_kernel_primitive:NN	\pdffontexpand	\pdfTEX_fontexpand:D
759	_kernel_primitive:NN	\pdffontsize	\pdfTEX_fontsize:D
760	_kernel_primitive:NN	\pdfignoreddimen	\pdfTEX_ignoreddimen:D
761	_kernel_primitive:NN	\pdfinsertht	\pdfTEX_insertht:D
762	_kernel_primitive:NN	\pdflastlinedepth	\pdfTEX_lastlinedepth:D
763	_kernel_primitive:NN	\pdflastxpos	\pdfTEX_lastxpos:D
764	_kernel_primitive:NN	\pdflastypos	\pdfTEX_lastypos:D
765	_kernel_primitive:NN	\pdfmapfile	\pdfTEX_mapfile:D
766	_kernel_primitive:NN	\pdfmapline	\pdfTEX_mapline:D
767	_kernel_primitive:NN	\pdfmdfivesum	\pdfTEX_mdfivesum:D
768	_kernel_primitive:NN	\pdfnoligatures	\pdfTEX_noligatures:D
769	_kernel_primitive:NN	\pdfnormaldeviate	\pdfTEX_normaldeviate:D
770	_kernel_primitive:NN	\pdfpageheight	\pdfTEX_pageheight:D
771	_kernel_primitive:NN	\pdfpagewidth	\pdfTEX_pagewidth:D
772	_kernel_primitive:NN	\pdfpkmode	\pdfTEX_pkmode:D
773	_kernel_primitive:NN	\pdfpkresolution	\pdfTEX_pkresolution:D
774	_kernel_primitive:NN	\pdfprimitive	\pdfTEX_primitive:D
775	_kernel_primitive:NN	\pdfprotrudechars	\pdfTEX_protrudechars:D
776	_kernel_primitive:NN	\pdfpxdimen	\pdfTEX_pxdimen:D
777	_kernel_primitive:NN	\pdfrandomseed	\pdfTEX_randomseed:D
778	_kernel_primitive:NN	\pdfsavepos	\pdfTEX_savepos:D
779	_kernel_primitive:NN	\pdfstrcmp	\pdfTEX_strcmp:D
780	_kernel_primitive:NN	\pdfsetrandomseed	\pdfTEX_setrandomseed:D
781	_kernel_primitive:NN	\pdfshellescape	\pdfTEX_shellescape:D
782	_kernel_primitive:NN	\pdftracingfonts	\pdfTEX_tracingfonts:D
783	_kernel_primitive:NN	\pdfuniformdeviate	\pdfTEX_uniformdeviate:D

The version primitives are not related to PDF mode but are related to pdfTEX so retain

the full prefix.

```

784 \__kernel_primitive:NN \pdfTeXbanner          \pdfTeXpdfTeXbanner:D
785 \__kernel_primitive:NN \pdfTeXrevision         \pdfTeXpdfTeXrevision:D
786 \__kernel_primitive:NN \pdfTeXversion          \pdfTeXpdfTeXversion:D

```

These ones appear in pdfTeX but don't have pdf in the name at all. (\synctex is odd as it's really not from pdfTeX but from SyncTeX!)

```

787 \__kernel_primitive:NN \efcode                 \pdfTeXefcode:D
788 \__kernel_primitive:NN \ifincsname             \pdfTeXifincsname:D
789 \__kernel_primitive:NN \leftmarginkern          \pdfTeXleftmarginkern:D
790 \__kernel_primitive:NN \letterspacefont         \pdfTeXletterspacefont:D
791 \__kernel_primitive:NN \lpcode                  \pdfTeXlpcode:D
792 \__kernel_primitive:NN \quitvmode               \pdfTeXquitvmode:D
793 \__kernel_primitive:NN \rightmarginkern          \pdfTeXrightmarginkern:D
794 \__kernel_primitive:NN \rpcode                  \pdfTeXrpcode:D
795 \__kernel_primitive:NN \synctex                 \pdfTeXsynctex:D
796 \__kernel_primitive:NN \tagcode                 \pdfTeXtagcode:D

```

Post pdfTeX primitive availability gets more complex. Both XeTeX and LuaTeX have varying names for some primitives from pdfTeX. Particularly for LuaTeX tracking all of that would be hard. Instead, we now check that we only save primitives if they actually exist.

```

797 </initex | names | package>
798 < *initex | package>
799 \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
800 \tex_long:D \tex_def:D \use_none:n #1 { }
801 \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
802 {
803   \etex_ifdefined:D #1
804   \tex_expandafter:D \use_ii:nn
805   \tex_fi:D
806   \use_none:n { \tex_global:D \tex_let:D #2 #1 }
807 < *initex>
808   \tex_global:D \tex_let:D #1 \tex_undefined:D
809 </initex>
810 }
811 </initex | package>
812 < *initex | names | package>

```

XeTeX-specific primitives. Note that XeTeX's \strcmp is handled earlier and is “rolled up” into \pdfstrcmp. With the exception of the version primitives these don't carry XeTeX through into the “base” name. A few cross-compatibility names which lack the pdf of the original are handled later.

```

813 \__kernel_primitive:NN \suppressfontnotfounderror
814 \xetex_suppressfontnotfounderror:D
815 \__kernel_primitive:NN \XeTeXcharclass          \xetex_charclass:D
816 \__kernel_primitive:NN \XeTeXcharglyph          \xetex_charglyph:D
817 \__kernel_primitive:NN \XeTeXcountfeatures        \xetex_countfeatures:D
818 \__kernel_primitive:NN \XeTeXcountglyphs          \xetex_countglyphs:D
819 \__kernel_primitive:NN \XeTeXcountselectors        \xetex_countselectors:D
820 \__kernel_primitive:NN \XeTeXcountvariations        \xetex_countvariations:D
821 \__kernel_primitive:NN \XeTeXdefaultencoding        \xetex_defaultencoding:D
822 \__kernel_primitive:NN \XeTeXdashbreakstate        \xetex_dashbreakstate:D
823 \__kernel_primitive:NN \XeTeXfeaturecode          \xetex_featurecode:D

```

```

824 \__kernel_primitive:NN \XeTeXfeaturename \xetex_featurename:D
825 \__kernel_primitive:NN \XeTeXfindfeaturebyname
826 \xetex_findfeaturebyname:D
827 \__kernel_primitive:NN \XeTeXfindselectorbyname
828 \xetex_findselectorbyname:D
829 \__kernel_primitive:NN \XeTeXfindvariationbyname
830 \xetex_findvariationbyname:D
831 \__kernel_primitive:NN \XeTeXfirstfontchar \xetex_firstfontchar:D
832 \__kernel_primitive:NN \XeTeXfonttype \xetex_fonttype:D
833 \__kernel_primitive:NN \XeTeXgenerateactualtext
834 \xetex_generateactualtext:D
835 \__kernel_primitive:NN \XeTeXglyph \xetex_glyph:D
836 \__kernel_primitive:NN \XeTeXglyphbounds \xetex_glyphbounds:D
837 \__kernel_primitive:NN \XeTeXglyphindex \xetex_glyphindex:D
838 \__kernel_primitive:NN \XeTeXglyphname \xetex_glyphname:D
839 \__kernel_primitive:NN \XeTeXinputencoding \xetex_inputencoding:D
840 \__kernel_primitive:NN \XeTeXinputnormalization
841 \xetex_inputnormalization:D
842 \__kernel_primitive:NN \XeTeXinterchartokenstate
843 \xetex_interchartokenstate:D
844 \__kernel_primitive:NN \XeTeXinterchartoks \xetex_interchartoks:D
845 \__kernel_primitive:NN \XeTeXisdefaultselector
846 \xetex_isdefaultselector:D
847 \__kernel_primitive:NN \XeTeXisexclusivefeature
848 \xetex_isexclusivefeature:D
849 \__kernel_primitive:NN \XeTeXlastfontchar \xetex_lastfontchar:D
850 \__kernel_primitive:NN \XeTeXlinebreakskip \xetex_linebreakskip:D
851 \__kernel_primitive:NN \XeTeXlinebreaklocale \xetex_linebreaklocale:D
852 \__kernel_primitive:NN \XeTeXlinebreakpenalty \xetex_linebreakpenalty:D
853 \__kernel_primitive:NN \XeTeXOTcountfeatures \xetex_OTcountfeatures:D
854 \__kernel_primitive:NN \XeTeXOTcountlanguages \xetex_OTcountlanguages:D
855 \__kernel_primitive:NN \XeTeXOTcountscripts \xetex_OTcountscripts:D
856 \__kernel_primitive:NN \XeTeXOTfeaturetag \xetex_OTfeaturetag:D
857 \__kernel_primitive:NN \XeTeXOTlanguagetag \xetex_OTlanguagetag:D
858 \__kernel_primitive:NN \XeTeXOTscripttag \xetex_OTscripttag:D
859 \__kernel_primitive:NN \XeTeXpdf file \xetex_pdf file:D
860 \__kernel_primitive:NN \XeTeXpdfpagecount \xetex_pdfpagecount:D
861 \__kernel_primitive:NN \XeTeXpicfile \xetex_picfile:D
862 \__kernel_primitive:NN \XeTeXselectorname \xetex_selectorname:D
863 \__kernel_primitive:NN \XeTeXtracingfonts \xetex_tracingfonts:D
864 \__kernel_primitive:NN \XeTeXupwardsmode \xetex_upwardsmode:D
865 \__kernel_primitive:NN \XeTeXuseglyphmetrics \xetex_useglyphmetrics:D
866 \__kernel_primitive:NN \XeTeXvariation \xetex_variation:D
867 \__kernel_primitive:NN \XeTeXvariationdefault \xetex_variationdefault:D
868 \__kernel_primitive:NN \XeTeXvariationmax \xetex_variationmax:D
869 \__kernel_primitive:NN \XeTeXvariationmin \xetex_variationmin:D
870 \__kernel_primitive:NN \XeTeXvariationname \xetex_variationname:D

```

The version primitives retain XeTeX.

```

871 \__kernel_primitive:NN \XeTeXrevision \xetex_XeTeXrevision:D
872 \__kernel_primitive:NN \XeTeXversion \xetex_XeTeXversion:D

```

Primitives from pdfTeX that XeTeX renames: also helps with LuaTeX.

```

873 \__kernel_primitive:NN \mdfivesum \pdf tex_mdfivesum:D
874 \__kernel_primitive:NN \ifprimitive \pdf tex_ifprimitive:D

```

875	<code>__kernel_primitive:NN \primitive</code>	<code>\pdfTeX_primitive:D</code>
876	<code>__kernel_primitive:NN \shellescape</code>	<code>\pdfTeX_shellescape:D</code>

Primitives from LuaT_EX, some of which have been ported back to X_YT_EX. Notice that `\expanded` was intended for pdfT_EX 1.50 but as that was not released we call this a LuaT_EX primitive.

877	<code>__kernel_primitive:NN \alignmark</code>	<code>\luatex_alignmark:D</code>
878	<code>__kernel_primitive:NN \alignstab</code>	<code>\luatex_alignstab:D</code>
879	<code>__kernel_primitive:NN \attribute</code>	<code>\luatex_attribute:D</code>
880	<code>__kernel_primitive:NN \attributedef</code>	<code>\luatex_attributedef:D</code>
881	<code>__kernel_primitive:NN \automaticdiscretionary</code>	
882	<code>\luatex_automaticdiscretionary:D</code>	
883	<code>__kernel_primitive:NN \automatichyphenmode</code>	
884	<code>\luatex_automatichyphenmode:D</code>	
885	<code>__kernel_primitive:NN \automatichyphenpenalty</code>	
886	<code>\luatex_automatichyphenpenalty:D</code>	
887	<code>__kernel_primitive:NN \beginscename</code>	<code>\luatex_beginscename:D</code>
888	<code>__kernel_primitive:NN \breakafterdirmode</code>	<code>\luatex_breakafterdirmode:D</code>
889	<code>__kernel_primitive:NN \catcodetable</code>	<code>\luatex_catcodetable:D</code>
890	<code>__kernel_primitive:NN \clearmarks</code>	<code>\luatex_clearmarks:D</code>
891	<code>__kernel_primitive:NN \crampeddisplaystyle</code>	
892	<code>\luatex_crampeddisplaystyle:D</code>	
893	<code>__kernel_primitive:NN \crampedscriptscriptstyle</code>	
894	<code>\luatex_crampedscriptscriptstyle:D</code>	
895	<code>__kernel_primitive:NN \crampedscriptstyle</code>	<code>\luatex_crampedscriptstyle:D</code>
896	<code>__kernel_primitive:NN \crampedtextstyle</code>	<code>\luatex_crampedtextstyle:D</code>
897	<code>__kernel_primitive:NN \directlua</code>	<code>\luatex_directlua:D</code>
898	<code>__kernel_primitive:NN \dviextension</code>	<code>\luatex_dviextension:D</code>
899	<code>__kernel_primitive:NN \dvifedback</code>	<code>\luatex_dvifedback:D</code>
900	<code>__kernel_primitive:NN \dvivariable</code>	<code>\luatex_dvivariable:D</code>
901	<code>__kernel_primitive:NN \etoksapp</code>	<code>\luatex_etoksapp:D</code>
902	<code>__kernel_primitive:NN \etokspre</code>	<code>\luatex_etokspre:D</code>
903	<code>__kernel_primitive:NN \explicithyphenpenalty</code>	
904	<code>\luatex_explicithyphenpenalty:D</code>	
905	<code>__kernel_primitive:NN \expanded</code>	<code>\luatex_expanded:D</code>
906	<code>__kernel_primitive:NN \explicitdiscretionary</code>	
907	<code>\luatex_explicitdiscretionary:D</code>	
908	<code>__kernel_primitive:NN \firstvalidlanguage</code>	<code>\luatex_firstvalidlanguage:D</code>
909	<code>__kernel_primitive:NN \fontid</code>	<code>\luatex_fontid:D</code>
910	<code>__kernel_primitive:NN \formatname</code>	<code>\luatex_formatname:D</code>
911	<code>__kernel_primitive:NN \hjcode</code>	<code>\luatex_hjcode:D</code>
912	<code>__kernel_primitive:NN \hpack</code>	<code>\luatex_hpack:D</code>
913	<code>__kernel_primitive:NN \hyphenationbounds</code>	<code>\luatex_hyphenationbounds:D</code>
914	<code>__kernel_primitive:NN \hyphenationmin</code>	<code>\luatex_hyphenationmin:D</code>
915	<code>__kernel_primitive:NN \hyphenpenaltymode</code>	<code>\luatex_hyphenpenaltymode:D</code>
916	<code>__kernel_primitive:NN \gladers</code>	<code>\luatex_gladers:D</code>
917	<code>__kernel_primitive:NN \initcatcodetable</code>	<code>\luatex_initcatcodetable:D</code>
918	<code>__kernel_primitive:NN \lastnamedcs</code>	<code>\luatex_lastnamedcs:D</code>
919	<code>__kernel_primitive:NN \latelua</code>	<code>\luatex_latelua:D</code>
920	<code>__kernel_primitive:NN \letcharcode</code>	<code>\luatex_letcharcode:D</code>
921	<code>__kernel_primitive:NN \luaescapestring</code>	<code>\luatex_luaescapestring:D</code>
922	<code>__kernel_primitive:NN \luafunction</code>	<code>\luatex_luafunction:D</code>
923	<code>__kernel_primitive:NN \luatexbanner</code>	<code>\luatex_luatexbanner:D</code>
924	<code>__kernel_primitive:NN \luatexrevision</code>	<code>\luatex_luatexrevision:D</code>

925	<code>__kernel_primitive:NN \luatexversion</code>	<code>\luatex luatexversion:D</code>
926	<code>__kernel_primitive:NN \mathdelimitersmode</code>	<code>\luatex_mathdelimitersmode:D</code>
927	<code>__kernel_primitive:NN \mathdisplayskipmode</code>	
928	<code>\luatex_mathdisplayskipmode:D</code>	
929	<code>__kernel_primitive:NN \matheqnogapstep</code>	<code>\luatex_matheqnogapstep:D</code>
930	<code>__kernel_primitive:NN \mathnolimitsmode</code>	<code>\luatex_mathnolimitsmode:D</code>
931	<code>__kernel_primitive:NN \mathoption</code>	<code>\luatex_mathoption:D</code>
932	<code>__kernel_primitive:NN \mathpenaltiesmode</code>	<code>\luatex_mathpenaltiesmode:D</code>
933	<code>__kernel_primitive:NN \mathrulesfam</code>	<code>\luatex_mathrulesfam:D</code>
934	<code>__kernel_primitive:NN \mathscriptsmode</code>	<code>\luatex_mathscriptsmode:D</code>
935	<code>__kernel_primitive:NN \mathscriptboxmode</code>	<code>\luatex_mathscriptboxmode:D</code>
936	<code>__kernel_primitive:NN \mathstyle</code>	<code>\luatex_mathstyle:D</code>
937	<code>__kernel_primitive:NN \mathsurroundmode</code>	<code>\luatex_mathsurroundmode:D</code>
938	<code>__kernel_primitive:NN \mathsurroundskip</code>	<code>\luatex_mathsurroundskip:D</code>
939	<code>__kernel_primitive:NN \nohrule</code>	<code>\luatex_nohrule:D</code>
940	<code>__kernel_primitive:NN \nokerns</code>	<code>\luatex_nokerns:D</code>
941	<code>__kernel_primitive:NN \noligs</code>	<code>\luatex_noligs:D</code>
942	<code>__kernel_primitive:NN \nospaces</code>	<code>\luatex_nospaces:D</code>
943	<code>__kernel_primitive:NN \novrule</code>	<code>\luatex_novrule:D</code>
944	<code>__kernel_primitive:NN \outputbox</code>	<code>\luatex_outputbox:D</code>
945	<code>__kernel_primitive:NN \pagebottomoffset</code>	<code>\luatex_pagebottomoffset:D</code>
946	<code>__kernel_primitive:NN \pageleftoffset</code>	<code>\luatex_pageleftoffset:D</code>
947	<code>__kernel_primitive:NN \pagerightoffset</code>	<code>\luatex_pagerightoffset:D</code>
948	<code>__kernel_primitive:NN \pagetopoffset</code>	<code>\luatex_pagetopoffset:D</code>
949	<code>__kernel_primitive:NN \pdfextension</code>	<code>\luatex_pdfextension:D</code>
950	<code>__kernel_primitive:NN \pdffeedback</code>	<code>\luatex_pdffeedback:D</code>
951	<code>__kernel_primitive:NN \pdfvariable</code>	<code>\luatex_pdfvariable:D</code>
952	<code>__kernel_primitive:NN \postexhyphenchar</code>	<code>\luatex_postexhyphenchar:D</code>
953	<code>__kernel_primitive:NN \posthyphenchar</code>	<code>\luatex_posthyphenchar:D</code>
954	<code>__kernel_primitive:NN \prebinoppenalty</code>	<code>\luatex_prebinoppenalty:D</code>
955	<code>__kernel_primitive:NN \predisplaygapfactor</code>	
956	<code>\luatex_predisplaygapfactor:D</code>	
957	<code>__kernel_primitive:NN \preexhyphenchar</code>	<code>\luatex_preexhyphenchar:D</code>
958	<code>__kernel_primitive:NN \prehyphenchar</code>	<code>\luatex_prehyphenchar:D</code>
959	<code>__kernel_primitive:NN \prerelpenalty</code>	<code>\luatex_prerelpenalty:D</code>
960	<code>__kernel_primitive:NN \savecatcodetable</code>	<code>\luatex_savecatcodetable:D</code>
961	<code>__kernel_primitive:NN \scantextokens</code>	<code>\luatex_scantextokens:D</code>
962	<code>__kernel_primitive:NN \setfontid</code>	<code>\luatex_setfontid:D</code>
963	<code>__kernel_primitive:NN \shapemode</code>	<code>\luatex_shapemode:D</code>
964	<code>__kernel_primitive:NN \suppressifcsnameerror</code>	
965	<code>\luatex_suppressifcsnameerror:D</code>	
966	<code>__kernel_primitive:NN \suppresslongerror</code>	<code>\luatex_suppresslongerror:D</code>
967	<code>__kernel_primitive:NN \suppressmathparerror</code>	
968	<code>\luatex_suppressmathparerror:D</code>	
969	<code>__kernel_primitive:NN \suppressoutererror</code>	<code>\luatex_suppressoutererror:D</code>
970	<code>__kernel_primitive:NN \suppressprimitiveerror</code>	
971	<code>\luatex_suppressprimitiveerror:D</code>	
972	<code>__kernel_primitive:NN \toksapp</code>	<code>\luatex_toksapp:D</code>
973	<code>__kernel_primitive:NN \tokspre</code>	<code>\luatex_tokspre:D</code>
974	<code>__kernel_primitive:NN \tpack</code>	<code>\luatex_tpack:D</code>
975	<code>__kernel_primitive:NN \vpack</code>	<code>\luatex_vpack:D</code>

Slightly more awkward are the directional primitives in LuaTeX. These come from Omega/Aleph, but we do not support those engines and so it seems most sensible to treat them as LuaTeX primitives for prefix purposes. One here is “new” but fits into the

general set.

```

976 \__kernel_primitive:NN \bodydir          \luatex_bodydir:D
977 \__kernel_primitive:NN \boxdir           \luatex_boxdir:D
978 \__kernel_primitive:NN \leftghost        \luatex_leftghost:D
979 \__kernel_primitive:NN \linedir          \luatex_linedir:D
980 \__kernel_primitive:NN \localbrokenpenalty \luatex_localbrokenpenalty:D
981 \__kernel_primitive:NN \localinterlinepenalty
982   \luatex_localinterlinepenalty:D
983 \__kernel_primitive:NN \lcalleftbox       \luatex_lcalleftbox:D
984 \__kernel_primitive:NN \lcalrightbox      \luatex_lcalrightbox:D
985 \__kernel_primitive:NN \mathdir          \luatex_mathdir:D
986 \__kernel_primitive:NN \pagedir          \luatex_pagedir:D
987 \__kernel_primitive:NN \pardir          \luatex_pardir:D
988 \__kernel_primitive:NN \rightghost       \luatex_rightghost:D
989 \__kernel_primitive:NN \textdir          \luatex_textdir:D

```

Primitives from pdfTeX that LuaTeX renames.

```

990 \__kernel_primitive:NN \adjustspacing     \pdfTeX_adjustspacing:D
991 \__kernel_primitive:NN \copyfont         \pdfTeX_copyfont:D
992 \__kernel_primitive:NN \draftmode        \pdfTeX_draftmode:D
993 \__kernel_primitive:NN \expandglyphsinfont \pdfTeX_fontexpand:D
994 \__kernel_primitive:NN \ifabsdim         \pdfTeX_ifabsdim:D
995 \__kernel_primitive:NN \ifabsnum         \pdfTeX_ifabsnum:D
996 \__kernel_primitive:NN \ignoreligaturesinfont
997   \pdfTeX_ignoreligaturesinfont:D
998 \__kernel_primitive:NN \insertht         \pdfTeX_insertht:D
999 \__kernel_primitive:NN \lastsavedboxresourceindex
1000   \pdfTeX_pdflastxform:D
1001 \__kernel_primitive:NN \lastsavedimageresourceindex
1002   \pdfTeX_pdflastximage:D
1003 \__kernel_primitive:NN \lastsavedimageresourcepages
1004   \pdfTeX_pdflastximagepages:D
1005 \__kernel_primitive:NN \lastxpos         \pdfTeX_lastxpos:D
1006 \__kernel_primitive:NN \lastypos        \pdfTeX_lastypos:D
1007 \__kernel_primitive:NN \normaldeviate    \pdfTeX_normaldeviate:D
1008 \__kernel_primitive:NN \outputmode       \pdfTeX_pdfoutput:D
1009 \__kernel_primitive:NN \pageheight       \pdfTeX_pageheight:D
1010 \__kernel_primitive:NN \pagewidth        \pdfTeX_pagewidth:D
1011 \__kernel_primitive:NN \protrudechars    \pdfTeX_protrudechars:D
1012 \__kernel_primitive:NN \pxdimen         \pdfTeX_pxdimen:D
1013 \__kernel_primitive:NN \randomseed       \pdfTeX_randomseed:D
1014 \__kernel_primitive:NN \useboxresource    \pdfTeX_pdfrefxform:D
1015 \__kernel_primitive:NN \useimageresource \pdfTeX_pdfrefximage:D
1016 \__kernel_primitive:NN \savepos          \pdfTeX_savepos:D
1017 \__kernel_primitive:NN \saveboxresource   \pdfTeX_pdfxform:D
1018 \__kernel_primitive:NN \saveimageresource \pdfTeX_pdfximage:D
1019 \__kernel_primitive:NN \setrandomseed    \pdfTeX_setrandomseed:D
1020 \__kernel_primitive:NN \tracingfonts     \pdfTeX_tracingfonts:D
1021 \__kernel_primitive:NN \uniformdeviate   \pdfTeX_uniformdeviate:D

```

The set of Unicode math primitives were introduced by XeTeX and LuaTeX in a somewhat complex fashion: a few first as \XeTeX... which were then renamed with LuaTeX having a lot more. These names now all start \U... and mainly \Umath.... To keep things somewhat clear we therefore prefix all of these as \utex... (introduced by a Uni-

code T_EX engine) and drop \U(math) from the names. Where there is a related T_EX90 primitive or where it really seems required we keep the math part of the name.

```

1022 \__kernel_primitive:NN \Uchar \utex_char:D
1023 \__kernel_primitive:NN \Ucharcat \utex_charcat:D
1024 \__kernel_primitive:NN \Udelcode \utex_delcode:D
1025 \__kernel_primitive:NN \Udelcodenum \utex_delcodenum:D
1026 \__kernel_primitive:NN \Udelimiter \utex_delimiter:D
1027 \__kernel_primitive:NN \Udelimiterover \utex_delimiterover:D
1028 \__kernel_primitive:NN \Udelimiterunder \utex_delimiterunder:D
1029 \__kernel_primitive:NN \Uhextensible \utex_hextensible:D
1030 \__kernel_primitive:NN \Umathaccent \utex_mathaccent:D
1031 \__kernel_primitive:NN \Umathaxis \utex_mathaxis:D
1032 \__kernel_primitive:NN \Umathbinbinspacing \utex_binbinspacing:D
1033 \__kernel_primitive:NN \Umathbinclosespacing \utex_binclosespacing:D
1034 \__kernel_primitive:NN \Umathbininnerspacing \utex_bininnerspacing:D
1035 \__kernel_primitive:NN \Umathbinopenspacing \utex_binopenspacing:D
1036 \__kernel_primitive:NN \Umathbinopspacing \utex_binopspacing:D
1037 \__kernel_primitive:NN \Umathbinordspacing \utex_binordspacing:D
1038 \__kernel_primitive:NN \Umathbinpunctspacing \utex_binpunctspacing:D
1039 \__kernel_primitive:NN \Umathbinrelspacing \utex_binrelspacing:D
1040 \__kernel_primitive:NN \Umathchar \utex_mathchar:D
1041 \__kernel_primitive:NN \Umathcharclass \utex_mathcharclass:D
1042 \__kernel_primitive:NN \Umathchardef \utex_mathchardef:D
1043 \__kernel_primitive:NN \Umathcharfam \utex_mathcharfam:D
1044 \__kernel_primitive:NN \Umathcharnum \utex_mathcharnum:D
1045 \__kernel_primitive:NN \Umathcharnumdef \utex_mathcharnumdef:D
1046 \__kernel_primitive:NN \Umathcharslot \utex_mathcharslot:D
1047 \__kernel_primitive:NN \Umathclosebinspacing \utex_closebinspacing:D
1048 \__kernel_primitive:NN \Umathcloseclosespacing
1049 \utex_closeclosespacing:D
1050 \__kernel_primitive:NN \Umathcloseinnerspacing
1051 \utex_closeinnerspacing:D
1052 \__kernel_primitive:NN \Umathcloseopenspacing \utex_closeopenspacing:D
1053 \__kernel_primitive:NN \Umathcloseopspacing \utex_closeopspacing:D
1054 \__kernel_primitive:NN \Umathcloseordspacing \utex_closeordspacing:D
1055 \__kernel_primitive:NN \Umathclosepunctspacing
1056 \utex_closepunctspacing:D
1057 \__kernel_primitive:NN \Umathcloserelspacing \utex_closerelspacing:D
1058 \__kernel_primitive:NN \Umathcode \utex_mathcode:D
1059 \__kernel_primitive:NN \Umathcodenum \utex_mathcodenum:D
1060 \__kernel_primitive:NN \Umathconnectoroverlapmin
1061 \utex_connectoroverlapmin:D
1062 \__kernel_primitive:NN \Umathfractiondelsize \utex_fractiondelsize:D
1063 \__kernel_primitive:NN \Umathfractiondenomdown
1064 \utex_fractiondenomdown:D
1065 \__kernel_primitive:NN \Umathfractiondenomvgap
1066 \utex_fractiondenomvgap:D
1067 \__kernel_primitive:NN \Umathfractionnumup \utex_fractionnumup:D
1068 \__kernel_primitive:NN \Umathfractionnumvgap \utex_fractionnumvgap:D
1069 \__kernel_primitive:NN \Umathfractionrule \utex_fractionrule:D
1070 \__kernel_primitive:NN \Umathinnerbinspacing \utex_innerbinspacing:D
1071 \__kernel_primitive:NN \Umathinnerclosespacing
1072 \utex_innerclosespacing:D
1073 \__kernel_primitive:NN \Umathinnerinnerspacing

```



```

1074 \utex_innerinnerspacing:D
1075 \__kernel_primitive:NN \Umathinneropenspacing \utex_inneropenspacing:D
1076 \__kernel_primitive:NN \Umathinneropspacing \utex_inneropspacing:D
1077 \__kernel_primitive:NN \Umathinnerordspacing \utex_innerordspacing:D
1078 \__kernel_primitive:NN \Umathinnerpunctspacing
1079 \utex_innerpunctspacing:D
1080 \__kernel_primitive:NN \Umathinnerrelspacing \utex_innerrelspacing:D
1081 \__kernel_primitive:NN \Umathlimitabovebgap \utex_limitabovebgap:D
1082 \__kernel_primitive:NN \Umathlimitabovekern \utex_limitabovekern:D
1083 \__kernel_primitive:NN \Umathlimitabovevgap \utex_limitabovevgap:D
1084 \__kernel_primitive:NN \Umathlimitbelowbgap \utex_limitbelowbgap:D
1085 \__kernel_primitive:NN \Umathlimitbelowkern \utex_limitbelowkern:D
1086 \__kernel_primitive:NN \Umathlimitbelowvgap \utex_limitbelowvgap:D
1087 \__kernel_primitive:NN \Umathnolimitsubfactor \utex_nolimitsubfactor:D
1088 \__kernel_primitive:NN \Umathnolimitsupfactor \utex_nolimitsupfactor:D
1089 \__kernel_primitive:NN \Umathopbinspacing \utex_opbinspacing:D
1090 \__kernel_primitive:NN \Umathopclosespacing \utex_opclosespacing:D
1091 \__kernel_primitive:NN \Umathopenbinspacing \utex_openbinspacing:D
1092 \__kernel_primitive:NN \Umathopenclosespacing \utex_openclosespacing:D
1093 \__kernel_primitive:NN \Umathopeninnerspacing \utex_openinnerspacing:D
1094 \__kernel_primitive:NN \Umathopenopenspacing \utex_openopenspacing:D
1095 \__kernel_primitive:NN \Umathopenopspacing \utex_openopspacing:D
1096 \__kernel_primitive:NN \Umathopenordspacing \utex_openordspacing:D
1097 \__kernel_primitive:NN \Umathopenpunctspacing \utex_openpunctspacing:D
1098 \__kernel_primitive:NN \Umathopenrelspacing \utex_openrelspacing:D
1099 \__kernel_primitive:NN \Umathoperatorsize \utex_operatorsize:D
1100 \__kernel_primitive:NN \Umathopinnerspacing \utex_opinnerspacing:D
1101 \__kernel_primitive:NN \Umathopopenspacing \utex_opopenspacing:D
1102 \__kernel_primitive:NN \Umathopopspacing \utex_opopspacing:D
1103 \__kernel_primitive:NN \Umathopordspacing \utex_opordspacing:D
1104 \__kernel_primitive:NN \Umathoppunctspacing \utex_oppunctspacing:D
1105 \__kernel_primitive:NN \Umathoprelspacing \utex_oprelspacing:D
1106 \__kernel_primitive:NN \Umathordbinspacing \utex_ordbinspacing:D
1107 \__kernel_primitive:NN \Umathordclosespacing \utex_ordclosespacing:D
1108 \__kernel_primitive:NN \Umathordinnerspacing \utex_ordinnerspacing:D
1109 \__kernel_primitive:NN \Umathordopenspacing \utex_ordopenspacing:D
1110 \__kernel_primitive:NN \Umathordopspacing \utex_ordopspacing:D
1111 \__kernel_primitive:NN \Umathordordspacing \utex_ordordspacing:D
1112 \__kernel_primitive:NN \Umathordpunctspacing \utex_ordpunctspacing:D
1113 \__kernel_primitive:NN \Umathordrelspacing \utex_ordrelspacing:D
1114 \__kernel_primitive:NN \Umathoverbarkern \utex_overbarkern:D
1115 \__kernel_primitive:NN \Umathoverbarrule \utex_overbarrule:D
1116 \__kernel_primitive:NN \Umathoverbarvgap \utex_overbarvgap:D
1117 \__kernel_primitive:NN \Umathoverdelimiterbgap
1118 \utex_overdelimiterbgap:D
1119 \__kernel_primitive:NN \Umathoverdelimitervgap
1120 \utex_overdelimitervgap:D
1121 \__kernel_primitive:NN \Umathpunctbinspacing \utex_punctbinspacing:D
1122 \__kernel_primitive:NN \Umathpunctclosespacing
1123 \utex_punctclosespacing:D
1124 \__kernel_primitive:NN \Umathpunctinnerspacing
1125 \utex_punctinnerspacing:D
1126 \__kernel_primitive:NN \Umathpunctopenspacing \utex_punctopenspacing:D
1127 \__kernel_primitive:NN \Umathpunctopspacing \utex_punctopspacing:D

```

1128	_kernel_primitive:NN	\Umathpunctordspacing	\utex_punctordspacing:D
1129	_kernel_primitive:NN	\Umathpunctpunctspacing	\utex_punctpunctspacing:D
1130	_kernel_primitive:NN	\Umathpunctrelspacing	\utex_punctrelspacing:D
1131	_kernel_primitive:NN	\Umathquad	\utex_quad:D
1132	_kernel_primitive:NN	\Umathradicaldegreeafter	
1133		\utex_radicaldegreeafter:D	
1134	_kernel_primitive:NN	\Umathradicaldegreebefore	
1135		\utex_radicaldegreebefore:D	
1136	_kernel_primitive:NN	\Umathradicaldegreeraise	
1137		\utex_radicaldegreeraise:D	
1138	_kernel_primitive:NN	\Umathradicalkern	\utex_radicalkern:D
1139	_kernel_primitive:NN	\Umathradicalrule	\utex_radicalrule:D
1140	_kernel_primitive:NN	\Umathradicalvgap	\utex_radicalvgap:D
1141	_kernel_primitive:NN	\Umathrelbinspacing	\utex_relbinspacing:D
1142	_kernel_primitive:NN	\Umathrelclosespacing	\utex_relclosespacing:D
1143	_kernel_primitive:NN	\Umathrelinnerspacing	\utex_relinnerspacing:D
1144	_kernel_primitive:NN	\Umathrelopenspacing	\utex_relopenspacing:D
1145	_kernel_primitive:NN	\Umathrellopspacing	\utex_rellopspacing:D
1146	_kernel_primitive:NN	\Umathrelordspacing	\utex_relordspacing:D
1147	_kernel_primitive:NN	\Umathrelpunctspacing	\utex_relpunctspacing:D
1148	_kernel_primitive:NN	\Umathrelrelspacing	\utex_relrelspacing:D
1149	_kernel_primitive:NN	\Umathskewedfractionhgap	
1150		\utex_skewedfractionhgap:D	
1151	_kernel_primitive:NN	\Umathskewedfractionvgap	
1152		\utex_skewedfractionvgap:D	
1153	_kernel_primitive:NN	\Umathspaceafterscript	\utex_spaceafterscript:D
1154	_kernel_primitive:NN	\Umathstackdenomdown	\utex_stackdenomdown:D
1155	_kernel_primitive:NN	\Umathstacknumup	\utex_stacknumup:D
1156	_kernel_primitive:NN	\Umathstackvgap	\utex_stackvgap:D
1157	_kernel_primitive:NN	\Umathsubshiftdown	\utex_subshiftdown:D
1158	_kernel_primitive:NN	\Umathsubshiftdrop	\utex_subshiftdrop:D
1159	_kernel_primitive:NN	\Umathsubsupshiftdown	\utex_subsupshiftdown:D
1160	_kernel_primitive:NN	\Umathsubsupvgap	\utex_subsupvgap:D
1161	_kernel_primitive:NN	\Umathsubtopmax	\utex_subtopmax:D
1162	_kernel_primitive:NN	\Umathsupbottommin	\utex_supbottommin:D
1163	_kernel_primitive:NN	\Umathsupshiftdrop	\utex_supshiftdrop:D
1164	_kernel_primitive:NN	\Umathsupshiftup	\utex_supshiftup:D
1165	_kernel_primitive:NN	\Umathsupsubbottommax	\utex_supsubbottommax:D
1166	_kernel_primitive:NN	\Umathunderbarkern	\utex_underbarkern:D
1167	_kernel_primitive:NN	\Umathunderbarrule	\utex_underbarrule:D
1168	_kernel_primitive:NN	\Umathunderbarvgap	\utex_underbarvgap:D
1169	_kernel_primitive:NN	\Umathunderdelimiterbgap	
1170		\utex_underdelimiterbgap:D	
1171	_kernel_primitive:NN	\Umathunderdelimitervgap	
1172		\utex_underdelimitervgap:D	
1173	_kernel_primitive:NN	\Unosubscript	\utex_nosubscript:D
1174	_kernel_primitive:NN	\Unosuperscript	\utex_nosuperscript:D
1175	_kernel_primitive:NN	\Uoverdelimiter	\utex_overdelimiter:D
1176	_kernel_primitive:NN	\Uradical	\utex_radical:D
1177	_kernel_primitive:NN	\Uroot	\utex_root:D
1178	_kernel_primitive:NN	\Uskewed	\utex_skewed:D
1179	_kernel_primitive:NN	\Uskewedwithdelims	\utex_skewedwithdelims:D
1180	_kernel_primitive:NN	\Ustack	\utex_stack:D
1181	_kernel_primitive:NN	\Ustartdisplaymath	\utex_startdisplaymath:D

1182	<code>__kernel_primitive:NN \Ustartmath</code>	<code>\utex_startmath:D</code>
1183	<code>__kernel_primitive:NN \Ustopdisplaymath</code>	<code>\utex_stopdisplaymath:D</code>
1184	<code>__kernel_primitive:NN \Ustopmath</code>	<code>\utex_stopmath:D</code>
1185	<code>__kernel_primitive:NN \Usubscript</code>	<code>\utex_subscript:D</code>
1186	<code>__kernel_primitive:NN \Usuperscript</code>	<code>\utex_superscript:D</code>
1187	<code>__kernel_primitive:NN \Uunderdelimit</code>	<code>\utex_underdelimit:D</code>
1188	<code>__kernel_primitive:NN \Uvextensible</code>	<code>\utex_vextensible:D</code>

Primitives from pL^AT_EX.

1189	<code>__kernel_primitive:NN \autospace</code>	<code>\ptex_autospace:D</code>
1190	<code>__kernel_primitive:NN \autoxspace</code>	<code>\ptex_autoxspace:D</code>
1191	<code>__kernel_primitive:NN \dtou</code>	<code>\ptex_dtou:D</code>
1192	<code>__kernel_primitive:NN \epTeXinputencoding</code>	<code>\ptex_inputencoding:D</code>
1193	<code>__kernel_primitive:NN \epTeXversion</code>	<code>\ptex_epTeXversion:D</code>
1194	<code>__kernel_primitive:NN \euc</code>	<code>\ptex_euc:D</code>
1195	<code>__kernel_primitive:NN \ifdbx</code>	<code>\ptex_ifdbx:D</code>
1196	<code>__kernel_primitive:NN \ifddir</code>	<code>\ptex_ifddir:D</code>
1197	<code>__kernel_primitive:NN \ifmdir</code>	<code>\ptex_ifmdir:D</code>
1198	<code>__kernel_primitive:NN \iftbx</code>	<code>\ptex_iftbx:D</code>
1199	<code>__kernel_primitive:NN \iftdir</code>	<code>\ptex_iftdir:D</code>
1200	<code>__kernel_primitive:NN \ifybx</code>	<code>\ptex_ifybx:D</code>
1201	<code>__kernel_primitive:NN \ifydir</code>	<code>\ptex_ifydir:D</code>
1202	<code>__kernel_primitive:NN \inhibitglue</code>	<code>\ptex_inhibitglue:D</code>
1203	<code>__kernel_primitive:NN \inhibitxspcode</code>	<code>\ptex_inhibitxspcode:D</code>
1204	<code>__kernel_primitive:NN \jcharwidowpenalty</code>	<code>\ptex_jcharwidowpenalty:D</code>
1205	<code>__kernel_primitive:NN \jfam</code>	<code>\ptex_jfam:D</code>
1206	<code>__kernel_primitive:NN \jfont</code>	<code>\ptex_jfont:D</code>
1207	<code>__kernel_primitive:NN \jis</code>	<code>\ptex_jis:D</code>
1208	<code>__kernel_primitive:NN \kanjiskip</code>	<code>\ptex_kanjiskip:D</code>
1209	<code>__kernel_primitive:NN \kansuji</code>	<code>\ptex_kansuji:D</code>
1210	<code>__kernel_primitive:NN \kansujichar</code>	<code>\ptex_kansujichar:D</code>
1211	<code>__kernel_primitive:NN \kcatcode</code>	<code>\ptex_kcatcode:D</code>
1212	<code>__kernel_primitive:NN \kuten</code>	<code>\ptex_kuten:D</code>
1213	<code>__kernel_primitive:NN \noautospace</code>	<code>\ptex_noautospace:D</code>
1214	<code>__kernel_primitive:NN \noautoxspace</code>	<code>\ptex_noautoxspace:D</code>
1215	<code>__kernel_primitive:NN \postbreakpenalty</code>	<code>\ptex_postbreakpenalty:D</code>
1216	<code>__kernel_primitive:NN \prebreakpenalty</code>	<code>\ptex_prebreakpenalty:D</code>
1217	<code>__kernel_primitive:NN \ptexversion</code>	<code>\ptex_ptexversion:D</code>
1218	<code>__kernel_primitive:NN \showmode</code>	<code>\ptex_showmode:D</code>
1219	<code>__kernel_primitive:NN \sjis</code>	<code>\ptex_sjis:D</code>
1220	<code>__kernel_primitive:NN \tate</code>	<code>\ptex_tate:D</code>
1221	<code>__kernel_primitive:NN \tbaselineshift</code>	<code>\ptex_tbaselineshift:D</code>
1222	<code>__kernel_primitive:NN \tfont</code>	<code>\ptex_tfont:D</code>
1223	<code>__kernel_primitive:NN \xkanjiskip</code>	<code>\ptex_xkanjiskip:D</code>
1224	<code>__kernel_primitive:NN \xspcode</code>	<code>\ptex_xspcode:D</code>
1225	<code>__kernel_primitive:NN \ybaselineshift</code>	<code>\ptex_ybaselineshift:D</code>
1226	<code>__kernel_primitive:NN \yoko</code>	<code>\ptex_yoko:D</code>

Primitives from upL^AT_EX.

1227	<code>__kernel_primitive:NN \disablecjktoken</code>	<code>\uptex_disablecjktoken:D</code>
1228	<code>__kernel_primitive:NN \enablecjktoken</code>	<code>\uptex_enablecjktoken:D</code>
1229	<code>__kernel_primitive:NN \forcecjktoken</code>	<code>\uptex_forcecjktoken:D</code>
1230	<code>__kernel_primitive:NN \kchar</code>	<code>\uptex_kchar:D</code>
1231	<code>__kernel_primitive:NN \kchardef</code>	<code>\uptex_kchardef:D</code>
1232	<code>__kernel_primitive:NN \kuten</code>	<code>\uptex_kuten:D</code>

```

1233 \__kernel_primitive:NN \ucs \uptex_ucs:D
1234 \__kernel_primitive:NN \uptexversion \uptex_uptexversion:D

```

End of the “just the names” part of the source.

```

1235 </initex | names | package>
1236 <*initex | package>

```

The job is done: close the group (using the primitive renamed!).

```

1237 \tex_endgroup:D

```

L^AT_ΕX 2_ε moves a few primitives, so these are sorted out. A convenient test for L^AT_ΕX 2_ε is the \@@end saved primitive.

```

1238 <*package>
1239 \etex_ifdefined:D \@@end
1240 \tex_let:D \tex_end:D \@@end
1241 \tex_let:D \tex_everydisplay:D \frozen@everydisplay
1242 \tex_let:D \tex_everymath:D \frozen@everymath
1243 \tex_let:D \tex_hyphen:D \@@hyph
1244 \tex_let:D \tex_input:D \@@input
1245 \tex_let:D \tex_italiccorrection:D \@@italiccorr
1246 \tex_let:D \tex_underline:D \@@underline

```

The \shipout primitive is particularly tricky as a number of packages want to hook in here. First, we see if a sufficiently-new kernel has saved a copy: if it has, just use that. Otherwise, we need to check each of the possible packages/classes that might move it: here, we are looking for those which do *not* delay action to the \AtBeginDocument hook. (We cannot use \primitive a (u)p_TΕX doesn’t offer it and as that doesn’t allow us to make a direct copy of the primitive *itself*.) As we know that L^AT_ΕX 2_ε is in use, we use it’s \@tfor loop here.

```

1247 \etex_ifdefined:D \@@shipout
1248 \tex_let:D \tex_shipout:D \@@shipout
1249 \tex_fi:D
1250 \tex_begingroup:D
1251 \tex_edef:D \l_tmpa_tl { \tex_string:D \shipout }
1252 \tex_edef:D \l_tmpb_tl { \tex_meaning:D \shipout }
1253 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1254 \tex_else:D
1255 \tex_expandafter:D \@tfor \tex_expandafter:D \@tempa \tex_string:D :=
1256 \CROP@shipout
1257 \dup@shipout
1258 \GPTorg@shipout
1259 \LL@shipout
1260 \mem@oldshipout
1261 \opem@shipout
1262 \pgfpages@originalshipout
1263 \pr@shipout
1264 \Shipout
1265 \verso@orig@shipout
1266 \do
1267 {
1268 \tex_edef:D \l_tmpb_tl
1269 { \tex_expandafter:D \tex_meaning:D \@tempa }
1270 \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
1271 \tex_global:D \tex_expandafter:D \tex_let:D
1272 \tex_expandafter:D \tex_shipout:D \@tempa

```

```

1273         \tex_fi:D
1274     }
1275     \tex_fi:D
1276 \tex_endgroup:D

```

Some tidying up is needed for `\(pdf)tracingfonts`. Newer LuaTeX has this simply as `\tracingfonts`, but that is overwritten by the L^AT_EX 2_ε kernel. So any spurious definition has to be removed, then the real version saved either from the pdfTeX name or from LuaTeX. In the latter case, we leave `\@@tracingfonts` available: this might be useful and almost all L^AT_EX 2_ε users will have `expl3` loaded by `fontspec`. (We follow the usual kernel convention that `@@` is used for saved primitives.)

```

1277 \tex_let:D \pdfTeX_tracingfonts:D \tex_undefined:D
1278 \etex_ifdefined:D \pdftracingfonts
1279 \tex_let:D \pdfTeX_tracingfonts:D \pdftracingfonts
1280 \tex_else:D
1281 \etex_ifdefined:D \luatex_directlua:D
1282 \luatex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
1283 \tex_let:D \pdfTeX_tracingfonts:D \luatextracingfonts
1284 \tex_fi:D
1285 \tex_fi:D
1286 \tex_fi:D

```

That is also true for the LuaTeX primitives under L^AT_EX 2_ε (depending on the format-building date). There are a few primitives that get the right names anyway so are missing here!

```

1287 \etex_ifdefined:D \luatexsuppressfontnotfounderror
1288 \tex_let:D \luatex_alignmark:D \luatexalignmark
1289 \tex_let:D \luatex_aligntab:D \luatexaligntab
1290 \tex_let:D \luatex_attribute:D \luatexattribute
1291 \tex_let:D \luatex_attributedef:D \luatexattributedef
1292 \tex_let:D \luatex_catcodetable:D \luatexcatcodetable
1293 \tex_let:D \luatex_clearmarks:D \luatexclearmarks
1294 \tex_let:D \luatex_crampeddisplaystyle:D \luatexcrampeddisplaystyle
1295 \tex_let:D \luatex_crampedscriptscriptstyle:D
1296 \luatexcrampedscriptscriptstyle
1297 \tex_let:D \luatex_crampedscriptstyle:D \luatexcrampedscriptstyle
1298 \tex_let:D \luatex_crampedtextstyle:D \luatexcrampedtextstyle
1299 \tex_let:D \luatex_fontid:D \luatexfontid
1300 \tex_let:D \luatex_formatname:D \luatexformatname
1301 \tex_let:D \luatex_gleaders:D \luatexgleaders
1302 \tex_let:D \luatex_initcatcodetable:D \luatexinitcatcodetable
1303 \tex_let:D \luatex_latelua:D \luatexlatelua
1304 \tex_let:D \luatex_luaescapestring:D \luatexluaescapestring
1305 \tex_let:D \luatex_luafunction:D \luatexluafunction
1306 \tex_let:D \luatex_mathstyle:D \luatexmathstyle
1307 \tex_let:D \luatex_nokerns:D \luatexnokerns
1308 \tex_let:D \luatex_noligs:D \luatexnoligs
1309 \tex_let:D \luatex_outputbox:D \luatexoutputbox
1310 \tex_let:D \luatex_pageleftoffset:D \luatexpageleftoffset
1311 \tex_let:D \luatex_pagetopoffset:D \luatexpagetopoffset
1312 \tex_let:D \luatex_postexhyphenchar:D \luatexpostexhyphenchar
1313 \tex_let:D \luatex_posthyphenchar:D \luatexposthyphenchar
1314 \tex_let:D \luatex_preexhyphenchar:D \luatexpreexhyphenchar
1315 \tex_let:D \luatex_prehyphenchar:D \luatexprehyphenchar

```

```

1316 \tex_let:D \luatex_savecatcodetable:D \luatexsavecatcodetable
1317 \tex_let:D \luatex_scantextokens:D \luatexscantextokens
1318 \tex_let:D \luatex_suppressifcsnameerror:D
1319 \luatexsuppressifcsnameerror
1320 \tex_let:D \luatex_suppresslongerror:D \luatexsuppresslongerror
1321 \tex_let:D \luatex_suppressmathparerror:D
1322 \luatexsuppressmathparerror
1323 \tex_let:D \luatex_suppressoutererror:D \luatexsuppressoutererror
1324 \tex_let:D \utex_char:D \luatexUchar
1325 \tex_let:D \xetex_suppressfontnotfounderror:D
1326 \luatexsuppressfontnotfounderror

```

Which also covers those slightly odd ones.

```

1327 \tex_let:D \luatex_bodydir:D \luatexbodydir
1328 \tex_let:D \luatex_boxdir:D \luatexboxdir
1329 \tex_let:D \luatex_leftghost:D \luatexleftghost
1330 \tex_let:D \luatex_localbrokenpenalty:D \luatexlocalbrokenpenalty
1331 \tex_let:D \luatex_localinterlinepenalty:D
1332 \luatexlocalinterlinepenalty
1333 \tex_let:D \luatex_localleftbox:D \luatexlocalleftbox
1334 \tex_let:D \luatex_localrightbox:D \luatexlocalrightbox
1335 \tex_let:D \luatex_mathdir:D \luatexmathdir
1336 \tex_let:D \luatex_pagebottomoffset:D \luatexpagebottomoffset
1337 \tex_let:D \luatex_pagedir:D \luatexpagedir
1338 \tex_let:D \pdfTeX_pageheight:D \luatexpageheight
1339 \tex_let:D \luatex_pagerightoffset:D \luatexpagerightoffset
1340 \tex_let:D \pdfTeX_pagewidth:D \luatexpagewidth
1341 \tex_let:D \luatex_pardir:D \luatexpardir
1342 \tex_let:D \luatex_rightghost:D \luatexrightghost
1343 \tex_let:D \luatex_textdir:D \luatextextdir
1344 \tex_fi:D

```

Only pdfTeX and LuaTeX define \pdfmapfile and \pdfmapline: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

```

1345 \tex_ifnum:D 0
1346 \etex_ifdefined:D \pdfTeX_pdfTeXversion:D 1 \tex_fi:D
1347 \etex_ifdefined:D \luatex luatexversion:D 1 \tex_fi:D
1348 = 0 %
1349 \tex_let:D \pdfTeX_mapfile:D \tex_undefined:D
1350 \tex_let:D \pdfTeX_mapline:D \tex_undefined:D
1351 \tex_fi:D
1352 </package>

```

Older XeTeX versions use \XeTeX as the prefix for the Unicode math primitives it knows. That is tidied up here (we support XeTeX versions from 0.9994 but this change was in 0.9999).

```

1353 (*initex | package)
1354 \etex_ifdefined:D \XeTeXdelcode
1355 \tex_let:D \utex_delcode:D \XeTeXdelcode
1356 \tex_let:D \utex_delcodenum:D \XeTeXdelcodenum
1357 \tex_let:D \utex_delimiter:D \XeTeXdelimiter
1358 \tex_let:D \utex_mathaccent:D \XeTeXmathaccent
1359 \tex_let:D \utex_mathchar:D \XeTeXmathchar
1360 \tex_let:D \utex_mathchardef:D \XeTeXmathchardef
1361 \tex_let:D \utex_mathcharnum:D \XeTeXmathcharnum

```

```

1362 \tex_let:D \utex_mathcharnumdef:D \XeTeXmathcharnumdef
1363 \tex_let:D \utex_mathcode:D \XeTeXmathcode
1364 \tex_let:D \utex_mathcodenum:D \XeTeXmathcodenum
1365 \tex_fi:D

```

Up to v0.80, LuaTeX defines the pdfTeX version data: rather confusing. Removing them means that `\pdfTeXpdfTeXversion:D` is a marker for pdfTeX alone: useful in engine-dependent code later.

```

1366 \etex_ifdefined:D \luatex luatexversion:D
1367 \tex_let:D \pdfTeXpdfTeXbanner:D \tex_undefined:D
1368 \tex_let:D \pdfTeXpdfTeXrevision:D \tex_undefined:D
1369 \tex_let:D \pdfTeXpdfTeXversion:D \tex_undefined:D
1370 \tex_fi:D
1371 </initex | package>

```

For ConTeXt, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using `\end` as a marker. For Mark IV, a few more primitives are moved: they are implemented using some Lua code in the current ConTeXt.

```

1372 <*package>
1373 \etex_ifdefined:D \normalend
1374 \tex_let:D \tex_end:D \normalend
1375 \tex_let:D \tex_everyjob:D \normaleveryjob
1376 \tex_let:D \tex_input:D \normalinput
1377 \tex_let:D \tex_language:D \normallanguage
1378 \tex_let:D \tex_mathop:D \normalmathop
1379 \tex_let:D \tex_month:D \normalmonth
1380 \tex_let:D \tex_outer:D \normalouter
1381 \tex_let:D \tex_over:D \normalover
1382 \tex_let:D \tex_vcenter:D \normalvcenter
1383 \tex_let:D \etex_unexpanded:D \normalunexpanded
1384 \tex_let:D \luatex_expanded:D \normalexpanded
1385 \tex_fi:D
1386 \etex_ifdefined:D \normalitaliccorrection
1387 \tex_let:D \tex_hoffset:D \normalhoffset
1388 \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
1389 \tex_let:D \tex_voffset:D \normalvoffset
1390 \tex_let:D \etex_showtokens:D \normalshowtokens
1391 \tex_let:D \luatex_bodydir:D \spac_directions_normal_body_dir
1392 \tex_let:D \luatex_pagedir:D \spac_directions_normal_page_dir
1393 \tex_fi:D
1394 \etex_ifdefined:D \normalleft
1395 \tex_let:D \tex_left:D \normalleft
1396 \tex_let:D \tex_middle:D \normalmiddle
1397 \tex_let:D \tex_right:D \normalright
1398 \tex_fi:D
1399 </package>
1400 </initex | package>

```

3 Internal kernel functions

<hr/> <code>_kernel_chk_cs_exist:N</code> <code>_kernel_chk_cs_exist:c</code> <hr/>	<code>_kernel_chk_cs_exist:N</code> $\langle cs \rangle$ This function is only created if debugging is enabled. It checks that $\langle cs \rangle$ exists according to the criteria for <code>\cs_if_exist_p:N</code> , and if not raises a kernel-level error.
<hr/> <code>_kernel_chk_defined:NT</code> <hr/>	<code>_kernel_chk_defined:NT</code> $\langle variable \rangle$ $\{ \langle true\ code \rangle \}$ If $\langle variable \rangle$ is not defined (according to <code>\cs_if_exist:NTF</code>), this triggers an error, otherwise the $\langle true\ code \rangle$ is run.
<hr/> <code>_kernel_chk_expr:nNn</code> <hr/>	<code>_kernel_chk_expr:nNn</code> $\{ \langle expr \rangle \}$ $\langle eval \rangle$ $\{ \langle convert \rangle \}$ $\langle caller \rangle$ This function is only created if debugging is enabled. By default it is equivalent to <code>\use_i:n</code> . When expression checking is enabled, it leaves in the input stream the result of <code>\tex_the:D</code> $\langle eval \rangle$ $\langle expr \rangle$ <code>\tex_relax:D</code> after checking that no token was left over. If any token was not taken as part of the expression, there is an error message displaying the result of the evaluation as well as the $\langle caller \rangle$. For instance $\langle eval \rangle$ can be <code>_int_eval:w</code> and $\langle caller \rangle$ can be <code>\int_eval:n</code> or <code>\int_set:Nn</code> . The argument $\langle convert \rangle$ is empty except for mu expressions where it is <code>\etex_mutoglue:D</code> , used for internal purposes.
<hr/> <code>_kernel_chk_var_exist:N</code> <hr/>	<code>_kernel_chk_var_exist:N</code> $\langle var \rangle$ This function is only created if debugging is enabled. It checks that $\langle var \rangle$ is defined according to the criteria for <code>\cs_if_exist_p:N</code> , and if not raises a kernel-level error.
<hr/> <code>_kernel_chk_var_scope:NN</code> <hr/>	<code>_kernel_chk_var_scope:NN</code> $\langle scope \rangle$ $\langle var \rangle$ Checks the $\langle var \rangle$ has the correct $\langle scope \rangle$, and if not raises a kernel-level error. This function is only created if debugging is enabled. The $\langle scope \rangle$ is a single letter <code>l</code> , <code>g</code> , <code>c</code> denoting local variables, global variables, or constants. More precisely, if the variable name starts with a letter and an underscore (normal <code>expl3</code> convention) the function checks that this single letter matches the $\langle scope \rangle$. Otherwise the function cannot know the scope $\langle var \rangle$ the first time: instead, it defines <code>_debug_chk_/\langle var\ name \rangle</code> to store that information for the next call. Thus, if a given $\langle var \rangle$ is subject to assignments of different scopes a kernel error will result.
<hr/> <code>_kernel_chk_var_local:N</code> <code>_kernel_chk_var_global:N</code> <hr/>	<code>_kernel_chk_var_local:N</code> $\langle var \rangle$ <code>_kernel_chk_var_global:N</code> $\langle var \rangle$ Applies <code>_kernel_chk_var_exist:N</code> $\langle var \rangle$, and assuming that is true applies <code>_kernel_chk_var_scope:NN</code> $\langle scope \rangle$ $\langle var \rangle$, where $\langle scope \rangle$ is <code>l</code> or <code>g</code> .
<hr/> <code>_kernel_cs_parm_from_arg_count:nnF</code> <hr/>	<code>_kernel_cs_parm_from_arg_count:nnF</code> $\{ \langle follow-on \rangle \}$ $\{ \langle args \rangle \}$ $\{ \langle false\ code \rangle \}$ Evaluates the number of $\langle args \rangle$ and leaves the $\langle follow-on \rangle$ code followed by a brace group containing the required number of primitive parameter markers (<code>#1</code> , <i>etc.</i>). If the number of $\langle args \rangle$ is outside the range $[0, 9]$, the $\langle false\ code \rangle$ is inserted <i>instead</i> of the $\langle follow-on \rangle$.

<hr/> <code>__kernel_if_debug:TF</code> <hr/>	<code>__kernel_if_debug:TF {<i><true code></i>} {<i><>false code></i>}</code>
	Runs the <i><true code></i> if debugging is enabled, namely only in L ^A T _E X 2 _ε package mode with one of the options <code>check-declarations</code> , <code>enable-debug</code> , or <code>log-functions</code> . Otherwise runs the <i><>false code></i> . The T and F variants are not provided for this low-level conditional.
<hr/> <code>__kernel_debug_log:x</code> <hr/>	<code>__kernel_debug_log:x {<i><message text></i>}</code>
	If the <code>log-functions</code> option is active, this function writes the <i><message text></i> to the log file using <code>\iow_log:x</code> . Otherwise, the <i><message text></i> is ignored using <code>\use_none:n</code> . This function is only created if debugging is enabled.
<hr/> <code>__kernel_exp_not:w *</code> <hr/>	<code>__kernel_exp_not:w <expandable tokens> {<i><content></i>}</code>
	Carries out expansion on the <i><expandable tokens></i> before preventing further expansion of the <i><content></i> as for <code>\exp_not:n</code> . Typically, the <i><expandable tokens></i> will alter the nature of the <i><content></i> , <i>i.e.</i> allow it to be generated in some way.
<code>\l__kernel_expl_bool</code>	A boolean which records the current code syntax status: <code>true</code> if currently inside a code environment. This variable should only be set by <code>\ExplSyntaxOn/\ExplSyntaxOff</code> . (End definition for <code>\l__kernel_expl_bool</code> .)
<hr/> <code>__kernel_file_missing:n</code> <hr/>	<code>__kernel_file_missing:n {<i><name></i>}</code>
	Expands the <i><name></i> as per <code>__kernel_file_name_sanitize:nN</code> then produces an error message indicating that that file was not found.
<hr/> <code>__kernel_file_name_sanitize:nN</code> <hr/>	<code>__kernel_file_name_sanitize:nN {<i><name></i>} <str var></code>
	For converting a <i><name></i> to a string where active characters are treated as strings.
<hr/> <code>__kernel_file_input_push:n</code> <code>__kernel_file_input_pop:</code> <hr/>	<code>__kernel_file_input_push:n {<i><name></i>}</code> <code>__kernel_file_input_pop:</code>
	Used to push and pop data from the internal file stack: needed only in package mode, where interfacing with the L ^A T _E X 2 _ε kernel is necessary.
<hr/> <code>__kernel_ior_open:Nn</code> <code>__kernel_ior_open:No</code> <hr/>	<code>__kernel_ior_open:Nn <stream> {<i><file name></i>}</code>
	This function has identical syntax to the public version. However, is does not take precautions against active characters in the <i><file name></i> , and it does not attempt to add a <i><path></i> to the <i><file name></i> : it is therefore intended to be used by higher-level functions which have already fully expanded the <i><file name></i> and which need to perform multiple open or close operations. See for example the implementation of <code>\file_get_full_name:nN</code> ,
<hr/> <code>__kernel_iow_with:Nnn</code> <hr/>	<code>__kernel_iow_with:Nnn <integer> {<i><value></i>} {<i><code></i>}</code>
	If the <i><integer></i> is equal to the <i><value></i> then this function simply runs the <i><code></i> . Otherwise it saves the current value of the <i><integer></i> , sets it to the <i><value></i> , runs the <i><code></i> , and restores the <i><integer></i> to its former value. This is used to ensure that the <code>\newlinechar</code> is 10 when writing to a stream, which lets <code>\iow_newline:</code> work, and that <code>\errorcontextlines</code> is -1 when displaying a message.

<hr/> <code>_kernel_msg_new:nnnn</code> <code>_kernel_msg_new:nnn</code> <hr/>	<code>_kernel_msg_new:nnnn {<module>} {<message>} {<text>} {<more text>}</code> <p>Creates a kernel <i><message></i> for a given <i><module></i>. The message is defined to first give <i><text></i> and then <i><more text></i> if the user requests it. If no <i><more text></i> is available then a standard text is given instead. Within <i><text></i> and <i><more text></i> four parameters (#1 to #4) can be used: these will be supplied and expanded at the time the message is used. An error is raised if the <i><message></i> already exists.</p>
<hr/> <code>_kernel_msg_set:nnnn</code> <code>_kernel_msg_set:nnn</code> <hr/>	<code>_kernel_msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}</code> <p>Sets up the text for a kernel <i><message></i> for a given <i><module></i>. The message is defined to first give <i><text></i> and then <i><more text></i> if the user requests it. If no <i><more text></i> is available then a standard text is given instead. Within <i><text></i> and <i><more text></i> four parameters (#1 to #4) can be used: these will be supplied and expanded at the time the message is used.</p>
<hr/> <code>_kernel_msg_fatal:nnnnnn</code> <code>_kernel_msg_fatal:nnxxxx</code> <code>_kernel_msg_fatal:nnnnnn</code> <code>_kernel_msg_fatal:nnxxx</code> <code>_kernel_msg_fatal:nnnn</code> <code>_kernel_msg_fatal:nnxx</code> <code>_kernel_msg_fatal:nnn</code> <code>_kernel_msg_fatal:nnx</code> <code>_kernel_msg_fatal:nn</code> <hr/>	<code>_kernel_msg_fatal:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code> <p>Issues kernel <i><module></i> error <i><message></i>, passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. After issuing a fatal error the T_EX run halts. Cannot be redirected.</p>
<hr/> <code>_kernel_msg_error:nnnnnn</code> <code>_kernel_msg_error:nnxxxx</code> <code>_kernel_msg_error:nnnnnn</code> <code>_kernel_msg_error:nnxxx</code> <code>_kernel_msg_error:nnnn</code> <code>_kernel_msg_error:nnxx</code> <code>_kernel_msg_error:nnn</code> <code>_kernel_msg_error:nnx</code> <code>_kernel_msg_error:nn</code> <hr/>	<code>_kernel_msg_error:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code> <p>Issues kernel <i><module></i> error <i><message></i>, passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The error stops processing and issues the text at the terminal. After user input, the run continues. Cannot be redirected.</p>
<hr/> <code>_kernel_msg_warning:nnnnnn</code> <code>_kernel_msg_warning:nnxxxx</code> <code>_kernel_msg_warning:nnnnnn</code> <code>_kernel_msg_warning:nnxxx</code> <code>_kernel_msg_warning:nnnn</code> <code>_kernel_msg_warning:nnxx</code> <code>_kernel_msg_warning:nnn</code> <code>_kernel_msg_warning:nnx</code> <code>_kernel_msg_warning:nn</code> <hr/>	<code>_kernel_msg_warning:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</code> <p>Issues kernel <i><module></i> warning <i><message></i>, passing <i><arg one></i> to <i><arg four></i> to the text-creating functions. The warning text is added to the log file, but the T_EX run is not interrupted.</p>

<code>__kernel_msg_info:nnnnnn</code>	<code>__kernel_msg_info:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg</code>
<code>__kernel_msg_info:nnxxxx</code>	<code>three>} {<arg four>}</code>
<code>__kernel_msg_info:nnnnn</code>	Issues kernel <i><module></i> information <i><message></i> , passing <i><arg one></i> to <i><arg four></i> to the
<code>__kernel_msg_info:nnxxx</code>	text-creating functions. The information text is added to the log file.
<code>__kernel_msg_info:nnnn</code>	
<code>__kernel_msg_info:nnxx</code>	
<code>__kernel_msg_info:nnn</code>	
<code>__kernel_msg_info:nnx</code>	
<code>__kernel_msg_info:nn</code>	

<code>__kernel_msg_expandable_error:nnnnnn</code>	<code>__kernel_msg_expandable_error:nnnnnn {<module>} {<message>}</code>
<code>__kernel_msg_expandable_error:nnffff</code>	<code>{<arg one>} {<arg two>} {<arg three>} {<arg four>}</code>
<code>__kernel_msg_expandable_error:nnnnn</code>	*
<code>__kernel_msg_expandable_error:nnfff</code>	*
<code>__kernel_msg_expandable_error:nnnn</code>	*
<code>__kernel_msg_expandable_error:nnff</code>	*
<code>__kernel_msg_expandable_error:nnn</code>	*
<code>__kernel_msg_expandable_error:nnf</code>	*
<code>__kernel_msg_expandable_error:nn</code>	*

Issues an error, passing *<arg one>* to *<arg four>* to the text-creating functions. The resulting string must be much shorter than a line, otherwise it is cropped.

<code>__kernel_patch:nnNNpn</code>	<code>__kernel_patch:nnNNpn {<before>} {<after>}</code>
	<code><definition> <function> <parameters> {<code>}</code>

If debugging is not enabled, this function ignores the *<before>* and *<after>* code and performs the *<definition>* with no patching. Otherwise it replaces *<code>* by *<before>* *<code>* *<after>* (which can involve #1 and so on) in the *<definition>* that follows. The *<definition>* must start with `\cs_new:Npn` or `\cs_set:Npn` or `\cs_gset:Npn` or their `_protected` counterparts. Other cases can be added as needed.

<code>__kernel_patch_conditional:nnNpnn</code>	<code>__kernel_patch_conditional:nnNpnn {<before>}</code>
	<code><definition> <conditional> <parameters> {<type>} {<code>}</code>

Similar to `__kernel_patch:nnNNpn` for conditionals, namely *<definition>* must be `\prg_new_conditional:Npnn` or its `_protected` counterpart. There is no *<after>* code because that would interfere with the action of the conditional.

<hr/> <code>__kernel_patch_args:nNNpn</code> <hr/>	<code>__kernel_patch_args:nNNpn {⟨arguments⟩}</code>
<code>__kernel_patch_conditional_args:nNNpnn</code>	<code>⟨definition⟩ ⟨function⟩ ⟨parameters⟩ {⟨code⟩}</code>

Like `__kernel_patch:nNNpn`, this tweaks the following definition, but from the “inside out” (and if debugging is not enabled, the `⟨arguments⟩` are ignored). It replaces `#1`, `#2` and so on in the `⟨code⟩` of the definition as indicated by the `⟨arguments⟩`. More precisely, a temporary function is defined using the `⟨definition⟩` with the `⟨parameters⟩` and `⟨code⟩`, then the result of expanding that function once in front of the `⟨arguments⟩` is used instead of the `⟨code⟩` when defining the actual function. For instance,

```
\__kernel_patch_args:nNNpn { { ( #1 ) } }
\cs_new:Npn \int_eval:n #1
{ \int_value:w \__int_eval:w #1 \__int_eval_end: }
```

would replace `#1` by `(#1)` in the definition of `\int_eval:n` when debugging is enabled. This fails if the `⟨code⟩` contains `##`. The `__kernel_patch_conditional_args:nNNpnn` function is for use before `\prg_new_conditional:Npn` or its `_protected` counterpart.

<hr/> <code>__kernel_patch_args:nnnNNpn</code> <hr/>	<code>__kernel_patch_args:nnnNNpn {⟨before⟩} {⟨after⟩}</code>
<code>__kernel_patch_conditional_args:nnnNNpnn</code>	<code>{⟨arguments⟩}</code> <code>⟨definition⟩ ⟨function⟩ ⟨parameters⟩ {⟨code⟩}</code>

A combination of `__kernel_patch:nNNpn` and `__kernel_patch_args:nNNpn`.

`\g__kernel_prg_map_int` This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions `\⟨type⟩_map_1:w`, `\⟨type⟩_map_2:w`, *etc.*, labelled by `\g__kernel_prg_map_int` hold functions to be mapped over various list datatypes in inline and variable mappings.

(End definition for `\g__kernel_prg_map_int`.)

<hr/> <code>__kernel_register_show:N</code> <hr/>	<code>__kernel_register_show:N ⟨register⟩</code>
<code>__kernel_register_show:c</code>	

Used to show the contents of a T_EX register at the terminal, formatted such that internal parts of the mechanism are not visible.

<hr/> <code>__kernel_register_log:N</code> <hr/>	<code>__kernel_register_log:N ⟨register⟩</code>
<code>__kernel_register_log:c</code>	

Used to write the contents of a T_EX register to the log file in a form similar to `__kernel_register_show:N`.

<hr/> <code>__kernel_str_to_other:n ★</code> <hr/>	<code>__kernel_str_to_other:n {⟨token list⟩}</code>
---	--

Converts the `⟨token list⟩` to a `⟨other string⟩`, where spaces have category code “other”. This function can be f-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string.

<hr/> <code>__kernel_str_to_other_fast:n ★</code> <hr/>	<code>__kernel_str_to_other_fast:n {⟨token list⟩}</code>
--	---

Same behaviour `__kernel_str_to_other:n` but only restricted-expandable. It takes a time linear in the character count of the string.

```
\_kernel_tl_to_str:w ★ \_kernel_tl_to_str:w <expandable tokens> {\tokens}
```

Carries out expansion on the $\langle expandable tokens \rangle$ before conversion of the $\langle tokens \rangle$ to a string as describe for $\backslash tl_to_str:n$. Typically, the $\langle expandable tokens \rangle$ will alter the nature of the $\langle tokens \rangle$, *i.e.* allow it to be generated in some way. This function requires only a single expansion.

4 l3basics implementation

```
1401 (*initex | package)
```

4.1 Renaming some T_EX primitives (again)

Having given all the T_EX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but we do a few now, just to get started.⁵

```
\if_true: Then some conditionals.
\if_false: 1402 \tex_let:D \if_true:      \tex_iftrue:D
\or:       1403 \tex_let:D \if_false:    \tex_iffalse:D
\else:     1404 \tex_let:D \or:      \tex_or:D
\fi:       1405 \tex_let:D \else:    \tex_else:D
\reverse_if:N 1406 \tex_let:D \fi:      \tex_fi:D
\if:w      1407 \tex_let:D \reverse_if:N \etex_unless:D
\if_charcode:w 1408 \tex_let:D \if:w      \tex_if:D
\if_catcode:w 1409 \tex_let:D \if_charcode:w \tex_if:D
\if_meaning:w 1410 \tex_let:D \if_catcode:w \tex_ifcat:D
1411 \tex_let:D \if_meaning:w \tex_ifx:D
```

(End definition for $\backslash if_true:$ and others. These functions are documented on page 20.)

```
\if_mode_math: TEX lets us detect some if its modes.
\if_mode_horizontal: 1412 \tex_let:D \if_mode_math:    \tex_ifmmode:D
\if_mode_vertical:   1413 \tex_let:D \if_mode_horizontal: \tex_ifhmode:D
\if_mode_inner:      1414 \tex_let:D \if_mode_vertical:  \tex_ifvmode:D
1415 \tex_let:D \if_mode_inner:    \tex_ifinner:D
```

(End definition for $\backslash if_mode_math:$ and others. These functions are documented on page 21.)

```
\if_cs_exist:N Building csnames and testing if control sequences exist.
\if_cs_exist:w 1416 \tex_let:D \if_cs_exist:N    \etex_ifdefined:D
\cs:w         1417 \tex_let:D \if_cs_exist:w    \etex_ifcsname:D
\cs_end:      1418 \tex_let:D \cs:w      \tex_csname:D
1419 \tex_let:D \cs_end:      \tex_endcsname:D
```

(End definition for $\backslash if_cs_exist:N$ and others. These functions are documented on page 21.)

```
\exp_after:wN The five \exp_ functions are used in the l3expan module where they are described.
\exp_not:N     1420 \tex_let:D \exp_after:wN    \tex_expandafter:D
\exp_not:n     1421 \tex_let:D \exp_not:N      \tex_noexpand:D
1422 \tex_let:D \exp_not:n      \etex_unexpanded:D
1423 \tex_let:D \exp:w         \tex_romannumeral:D
1424 \tex_chardef:D \exp_end: = 0 ~
```

⁵This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the $\backslash tex_...:D$ name in the cases where no good alternative exists.

(End definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 30.)

`\token_to_meaning:N` Examining a control sequence or token.

```
\cs_meaning:N 1425 \tex_let:D \token_to_meaning:N \tex_meaning:D
1426 \tex_let:D \cs_meaning:N \tex_meaning:D
```

(End definition for `\token_to_meaning:N` and `\cs_meaning:N`. These functions are documented on page 118.)

`\tl_to_str:n` Making strings.

```
\token_to_str:N 1427 \tex_let:D \tl_to_str:n \etex_detokenize:D
\__kernel_tl_to_str:w 1428 \tex_let:D \token_to_str:N \tex_string:D
1429 \tex_let:D \__kernel_tl_to_str:w \etex_detokenize:D
```

(End definition for `\tl_to_str:n`, `\token_to_str:N`, and `__kernel_tl_to_str:w`. These functions are documented on page 41.)

`\scan_stop:` The next three are basic functions for which there also exist versions that are safe inside alignments. These safe versions are defined in the `l3prg` module.

```
\group_begin: 1430 \tex_let:D \scan_stop: \tex_relax:D
1431 \tex_let:D \group_begin: \tex_begingroup:D
1432 \tex_let:D \group_end: \tex_endgroup:D
```

(End definition for `\scan_stop:`, `\group_begin:`, and `\group_end:`. These functions are documented on page 8.)

```
1433 <@@=int>
```

`\if_int_compare:w` For integers.

```
\__int_to_roman:w 1434 \tex_let:D \if_int_compare:w \tex_ifnum:D
1435 \tex_let:D \__int_to_roman:w \tex_romannumeral:D
```

(End definition for `\if_int_compare:w` and `__int_to_roman:w`. This function is documented on page 88.)

`\group_insert_after:N` Adding material after the end of a group.

```
1436 \tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(End definition for `\group_insert_after:N`. This function is documented on page 8.)

`\exp_args:Nc` Discussed in `l3expan`, but needed much earlier.

```
\exp_args:cc 1437 \tex_long:D \tex_def:D \exp_args:Nc #1#2
1438 { \exp_after:wN #1 \cs:w #2 \cs_end: }
1439 \tex_long:D \tex_def:D \exp_args:cc #1#2
1440 { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }
```

(End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 26.)

`\token_to_meaning:c` A small number of variants defined by hand. Some of the necessary functions (`\use_i:nn`, `\use_ii:nn`, and `\exp_args:Nnc`) are not defined at that point yet, but will be defined before those variants are used. The `\cs_meaning:c` command must check for an undefined control sequence to avoid defining it mistakenly.

```

1441 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
1442 \tex_long:D \tex_def:D \cs_meaning:c #1
1443 {
1444   \if_cs_exist:w #1 \cs_end:
1445     \exp_after:wN \use_i:nn
1446   \else:
1447     \exp_after:wN \use_ii:nn
1448   \fi:
1449   { \exp_args:Nc \cs_meaning:N {#1} }
1450   { \tl_to_str:n {undefined} }
1451 }
1452 \tex_let:D \token_to_meaning:c = \cs_meaning:c

```

(End definition for `\token_to_meaning:N`. This function is documented on page 118.)

4.2 Defining some constants

`\c_zero` We need the constant `\c_zero` which is used by some functions in the `l3alloc` module. The rest are defined in the `l3int` module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the `l3int` module is required but it can’t be used until the allocation has been set up properly!

```

1453 \tex_chardef:D \c_zero = 0 ~

```

(End definition for `\c_zero`. This variable is documented on page 87.)

`\c_max_register_int` This is here as this particular integer is needed both in package mode and to bootstrap `l3alloc`, and is documented in `l3int`.

```

1454 \etex_ifdefined:D \luatex luatexversion:D
1455 \tex_chardef:D \c_max_register_int = 65 535 ~
1456 \tex_else:D
1457 \tex_mathchardef:D \c_max_register_int = 32 767 ~
1458 \tex_fi:D

```

(End definition for `\c_max_register_int`. This variable is documented on page 87.)

4.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

`\cs_set_nopar:Npn` All assignment functions in L^AT_EX3 should be naturally protected; after all, the T_EX primitives for assignments are and it can be a cause of problems if others aren’t.

`\cs_set_nopar:Npx`

`\cs_set:Npn`

`\cs_set:Npx`

`\cs_set_protected_nopar:Npn`

`\cs_set_protected_nopar:Npx`

`\cs_set_protected:Npn`

`\cs_set_protected:Npx`

```

1459 \tex_let:D \cs_set_nopar:Npn \tex_def:D
1460 \tex_let:D \cs_set_nopar:Npx \tex_edef:D
1461 \etex_protected:D \tex_long:D \tex_def:D \cs_set:Npn
1462 { \tex_long:D \tex_def:D }
1463 \etex_protected:D \tex_long:D \tex_def:D \cs_set:Npx
1464 { \tex_long:D \tex_edef:D }
1465 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npn
1466 { \etex_protected:D \tex_def:D }

```

```

1467 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npx
1468 { \etex_protected:D \tex_edef:D }
1469 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npn
1470 { \etex_protected:D \tex_long:D \tex_def:D }
1471 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npx
1472 { \etex_protected:D \tex_long:D \tex_edef:D }

```

(End definition for `\cs_set_nopar:Npn` and others. These functions are documented on page 10.)

`\cs_gset_nopar:Npn` Global versions of the above functions.

```

\cs_gset_nopar:Npx 1473 \tex_let:D \cs_gset_nopar:Npn \tex_gdef:D
\cs_gset:Npn 1474 \tex_let:D \cs_gset_nopar:Npx \tex_xdef:D
\cs_gset:Npx 1475 \cs_set_protected:Npn \cs_gset:Npn
\cs_gset_protected_nopar:Npn 1476 { \tex_long:D \tex_gdef:D }
\cs_gset_protected_nopar:Npx 1477 \cs_set_protected:Npn \cs_gset:Npx
\cs_gset_protected:Npn 1478 { \tex_long:D \tex_xdef:D }
\cs_gset_protected:Npn 1479 \cs_set_protected:Npn \cs_gset_protected_nopar:Npn
\cs_gset_protected:Npx 1480 { \etex_protected:D \tex_gdef:D }
\cs_gset_protected:Npx 1481 \cs_set_protected:Npn \cs_gset_protected_nopar:Npx
\cs_gset_protected:Npx 1482 { \etex_protected:D \tex_xdef:D }
\cs_gset_protected:Npx 1483 \cs_set_protected:Npn \cs_gset_protected:Npn
\cs_gset_protected:Npx 1484 { \etex_protected:D \tex_long:D \tex_gdef:D }
\cs_gset_protected:Npx 1485 \cs_set_protected:Npn \cs_gset_protected:Npx
\cs_gset_protected:Npx 1486 { \etex_protected:D \tex_long:D \tex_xdef:D }

```

(End definition for `\cs_gset_nopar:Npn` and others. These functions are documented on page 11.)

4.4 Selecting tokens

```

1487 <@@=exp>

```

`\l__exp_internal_tl` Scratch token list variable for `\l3expan`, used by `\use:x`, used in defining conditionals. We don't use `tl` methods because `\l3basics` is loaded earlier.

```

1488 \cs_set_nopar:Npn \l__exp_internal_tl { }

```

(End definition for `\l__exp_internal_tl`.)

`\use:c` This macro grabs its argument and returns a csname from it.

```

1489 \cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }

```

(End definition for `\use:c`. This function is documented on page 15.)

`\use:x` Fully expands its argument and passes it to the input stream. Uses the reserved `\l__exp_internal_tl` which will be set up in `\l3expan`.

```

1490 \cs_set_protected:Npn \use:x #1
1491 {
1492   \cs_set_nopar:Npx \l__exp_internal_tl {#1}
1493   \l__exp_internal_tl
1494 }

```

(End definition for `\use:x`. This function is documented on page 18.)

```

1495 <@@=use>

```

`\use_x:n` A candidate, but needed early as it links to string comparisons. Currently LuaTeX-only.

```

1496 \cs_set:Npn \use_x:n #1 { \luatex_expanded:D {#1} }

```


(End definition for `\use_x:n`. This function is documented on page 235.)

```
1497 <@@=exp>
```

`\use:n` These macros grab their arguments and return them back to the input (with outer braces removed).
`\use:nn`

```
\use:nnn 1498 \cs_set:Npn \use:n #1 {#1}
\use:nnnn 1499 \cs_set:Npn \use:nn #1#2 {#1#2}
1500 \cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
1501 \cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

(End definition for `\use:n` and others. These functions are documented on page 17.)

`\use_i:nn` The equivalent to L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`.
`\use_ii:nn`

```
1502 \cs_set:Npn \use_i:nn #1#2 {#1}
1503 \cs_set:Npn \use_ii:nn #1#2 {#2}
```

(End definition for `\use_i:nn` and `\use_ii:nn`. These functions are documented on page 17.)

`\use_i:nnn` We also need something for picking up arguments from a longer list.
`\use_ii:nnn`

```
\use_iii:nnn 1504 \cs_set:Npn \use_i:nnn #1#2#3 {#1}
\use_iii:nnn 1505 \cs_set:Npn \use_ii:nnn #1#2#3 {#2}
\use_i_ii:nnn 1506 \cs_set:Npn \use_iii:nnn #1#2#3 {#3}
\use_i:nnnn 1507 \cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
\use_ii:nnnn 1508 \cs_set:Npn \use_i:nnnn #1#2#3#4 {#1}
\use_iii:nnnn 1509 \cs_set:Npn \use_ii:nnnn #1#2#3#4 {#2}
\use_iv:nnnn 1510 \cs_set:Npn \use_iii:nnnn #1#2#3#4 {#3}
1511 \cs_set:Npn \use_iv:nnnn #1#2#3#4 {#4}
```

(End definition for `\use_i:nnn` and others. These functions are documented on page 17.)

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.
`\use_none_delimit_by_q_stop:w`
`\use_none_delimit_by_q_recursion_stop:w`

```
1512 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
1513 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
1514 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
```

(End definition for `\use_none_delimit_by_q_nil:w`, `\use_none_delimit_by_q_stop:w`, and `\use_none_delimit_by_q_recursion_stop:w`. These functions are documented on page 18.)

`\use_i_delimit_by_q_nil:nw` Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.
`\use_i_delimit_by_q_stop:nw`
`\use_i_delimit_by_q_recursion_stop:nw`

```
1515 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
1516 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
1517 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw
1518 #1#2 \q_recursion_stop {#1}
```

(End definition for `\use_i_delimit_by_q_nil:nw`, `\use_i_delimit_by_q_stop:nw`, and `\use_i_delimit_by_q_recursion_stop:nw`. These functions are documented on page 18.)

4.5 Gobbling tokens from input

`\use_none:n` To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of n's following the : in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once takes care of gobbling all the tokens in one go.

```

1519 \cs_set:Npn \use_none:n      #1          { }
1520 \cs_set:Npn \use_none:nn     #1#2        { }
1521 \cs_set:Npn \use_none:nnn    #1#2#3      { }
1522 \cs_set:Npn \use_none:nnnn   #1#2#3#4    { }
1523 \cs_set:Npn \use_none:nnnnn  #1#2#3#4#5  { }
1524 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6 { }
1525 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 { }
1526 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 { }
1527 \cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 { }

```

(End definition for `\use_none:n` and others. These functions are documented on page 18.)

4.6 Debugging and patching later definitions

```

1528 <@@=debug>

```

`__kernel_if_debug:TF` A more meaningful test of whether debugging is enabled than messing up with guards. We can also more easily change the logic in one place then. At present, debugging is disabled in the format and in generic mode, while in L^AT_EX_ε mode it is enabled if one of the options `enable-debug`, `log-functions` or `check-declarations` was given.

```

1529 \cs_set_protected:Npn \__kernel_if_debug:TF #1#2 {#2}
1530 <*package>
1531 \tex_ifodd:D \l@expl@enable@debug@bool
1532 \cs_set_protected:Npn \__kernel_if_debug:TF #1#2 {#1}
1533 \fi:
1534 </package>

```

(End definition for `__kernel_if_debug:TF`.)

```

\debug_on:n
\debug_off:n
1535 \__kernel_if_debug:TF
1536 {
1537   \cs_set_protected:Npn \debug_on:n #1
1538   {
1539     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
1540     {
1541       \cs_if_exist_use:cF { __debug_ ##1 _on: }
1542       { \__kernel_msg_error:nnn { kernel } { debug } {##1} }
1543     }
1544   }
1545   \cs_set_protected:Npn \debug_off:n #1
1546   {
1547     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
1548     {
1549       \cs_if_exist_use:cF { __debug_ ##1 _off: }
1550       { \__kernel_msg_error:nnn { kernel } { debug } {##1} }
1551     }

```

```

1552     }
1553 }
1554 {
1555     \cs_set_protected:Npn \debug_on:n #1
1556     {
1557         \__kernel_msg_error:nnx { kernel } { enable-debug }
1558         { \tl_to_str:n { \debug_on:n {#1} } }
1559     }
1560     \cs_set_protected:Npn \debug_off:n #1
1561     {
1562         \__kernel_msg_error:nnx { kernel } { enable-debug }
1563         { \tl_to_str:n { \debug_off:n {#1} } }
1564     }
1565 }

```

(End definition for `\debug_on:n` and `\debug_off:n`. These functions are documented on page 235.)

`\debug_suspend:` Suspend and resume locally all debug-related errors and logging except deprecation errors.

`\debug_resume:` The `\debug_suspend:` and `\debug_resume:` pairs can be nested. We keep track of nesting in a token list containing a number of periods. At first begin with the “non-suspended” version of `__debug_suspended:T`.

`__debug_suspended:T`
`\l__debug_suspended_tl`

```

1566 \__kernel_if_debug:TF
1567 {
1568     \cs_set_nopar:Npn \l__debug_suspended_tl { }
1569     \cs_set_protected:Npn \debug_suspend:
1570     {
1571         \tl_put_right:Nn \l__debug_suspended_tl { . }
1572         \cs_set_eq:NN \__debug_suspended:T \use:n
1573     }
1574     \cs_set_protected:Npn \debug_resume:
1575     {
1576         \tl_set:Nx \l__debug_suspended_tl
1577         { \tl_tail:N \l__debug_suspended_tl }
1578         \tl_if_empty:NT \l__debug_suspended_tl
1579         {
1580             \cs_set_eq:NN \__debug_suspended:T \use_none:n
1581         }
1582     }
1583     \cs_set:Npn \__debug_suspended:T #1 { }
1584 }
1585 {
1586     \cs_set_protected:Npn \debug_suspend: { }
1587     \cs_set_protected:Npn \debug_resume: { }
1588 }

```

(End definition for `\debug_suspend:` and others. These functions are documented on page 235.)

When debugging is enabled these two functions set up functions that test their argument (when `check-declarations` is active)

`__debug_check-declarations_on:`
`__debug_check-declarations_off:`
`__kernel_chk_var_exist:N`
`__kernel_chk_cs_exist:N`
`__kernel_chk_cs_exist:c`
`__kernel_chk_var_local:N`
`__kernel_chk_var_global:N`
`__kernel_chk_var_scope:NN`

- `__kernel_chk_var_exist:N` and `__kernel_chk_cs_exist:N`, two functions that test that their argument is defined;
- `__kernel_chk_var_scope:NN` that checks that its argument #2 has scope #1.

- `__kernel_chk_var_local:N` and `__kernel_chk_var_global:N` that perform both checks.

```

1589 \__kernel_if_debug:TF
1590 {
1591   \exp_args:Nc \cs_set_protected:Npn { __debug_check-declarations_on: }
1592   {
1593     \cs_set_protected:Npn \__kernel_chk_var_exist:N ##1
1594     {
1595       \__debug_suspended:T \use_none:nnn
1596       \cs_if_exist:NF ##1
1597       {
1598         \__kernel_msg_error:nxx { kernel } { non-declared-variable }
1599         { \token_to_str:N ##1 }
1600       }
1601     }
1602     \cs_set_protected:Npn \__kernel_chk_cs_exist:N ##1
1603     {
1604       \__debug_suspended:T \use_none:nnn
1605       \cs_if_exist:NF ##1
1606       {
1607         \__kernel_msg_error:nxx { kernel } { command-not-defined }
1608         { \token_to_str:N ##1 }
1609       }
1610     }
1611     \cs_set_protected:Npn \__kernel_chk_var_scope:NN
1612     {
1613       \__debug_suspended:T \use_none:nnn
1614       \__debug_chk_var_scope_aux:NN
1615     }
1616     \cs_set_protected:Npn \__kernel_chk_var_local:N ##1
1617     {
1618       \__debug_suspended:T \use_none:nnnnn
1619       \__kernel_chk_var_exist:N ##1
1620       \__debug_chk_var_scope_aux:NN l ##1
1621     }
1622     \cs_set_protected:Npn \__kernel_chk_var_global:N ##1
1623     {
1624       \__debug_suspended:T \use_none:nnnnn
1625       \__kernel_chk_var_exist:N ##1
1626       \__debug_chk_var_scope_aux:NN g ##1
1627     }
1628   }
1629   \exp_args:Nc \cs_set_protected:Npn { __debug_check-declarations_off: }
1630   {
1631     \cs_set_protected:Npn \__kernel_chk_var_exist:N ##1 { }
1632     \cs_set_protected:Npn \__kernel_chk_cs_exist:N ##1 { }
1633     \cs_set_protected:Npn \__kernel_chk_var_local:N ##1 { }
1634     \cs_set_protected:Npn \__kernel_chk_var_global:N ##1 { }
1635     \cs_set_protected:Npn \__kernel_chk_var_scope:NN ##1##2 { }
1636   }
1637   \cs_set_protected:Npn \__kernel_chk_cs_exist:c
1638   { \exp_args:Nc \__kernel_chk_cs_exist:N }
1639   \tex_ifodd:D \l@expl@check@declarations@bool
1640   \use:c { __debug_check-declarations_on: }

```

```

1641     \else:
1642         \use:c { __debug_check-declarations_off: }
1643     \fi:
1644 }
1645 { }

```

(End definition for `__debug_check-declarations_on:` and others.)

```

\__debug_chk_var_scope_aux:NN
\__debug_chk_var_scope_aux:Nn
\__debug_chk_var_scope_aux:NNn

```

First check whether the name of the variable #2 starts with $\langle letter \rangle$ _. If it does then pass that letter, the $\langle scope \rangle$, and the variable name to `__debug_chk_var_scope_aux:NNn`. That function compares the two letters and triggers an error if they differ (the `\scan_stop:` case is not reachable here). If the second character was not _ then pass the same data to the same auxiliary, except for its first argument which is now a control sequence. That control sequence is actually a token list (but to avoid triggering the checking code we manipulate it using `\cs_set_nopar:Npn`) containing a single letter $\langle scope \rangle$ according to what the first assignment to the given variable was.

```

1646 \__kernel_if_debug:TF
1647 {
1648     \cs_set_protected:Npn \__debug_chk_var_scope_aux:NN #1#2
1649     { \exp_args:NNf \__debug_chk_var_scope_aux:Nn #1 { \cs_to_str:N #2 } }
1650     \cs_set_protected:Npn \__debug_chk_var_scope_aux:Nn #1#2
1651     {
1652         \if:w _ \use_i:nn \use_i_delimit_by_q_stop:nw #2 ? ? \q_stop
1653         \exp_after:wN \__debug_chk_var_scope_aux:NNn
1654         \use_i_delimit_by_q_stop:nw #2 ? \q_stop
1655         #1 {#2}
1656     \else:
1657         \exp_args:Nc \__debug_chk_var_scope_aux:NNn
1658         { __debug_chk_/ #2 }
1659         #1 {#2}
1660     \fi:
1661 }
1662 \cs_set_protected:Npn \__debug_chk_var_scope_aux:NNn #1#2#3
1663 {
1664     \if:w #1 #2
1665     \else:
1666         \if:w #1 \scan_stop:
1667             \cs_gset_nopar:Npn #1 {#2}
1668         \else:
1669             \__kernel_msg_error:nxxxx { kernel } { local-global }
1670             {#1} {#2} { \iow_char:N \ \ #3 }
1671         \fi:
1672     \fi:
1673 }
1674 }
1675 { }

```

(End definition for `__debug_chk_var_scope_aux:NN`, `__debug_chk_var_scope_aux:Nn`, and `__debug_chk_var_scope_aux:NNn`.)

```

\__debug_check-expressions_on:
\__debug_check-expressions_off:
\__kernel_chk_expr:nNnN
\__debug_chk_expr_aux:nNnN

```

When debugging is enabled these two functions set `__kernel_chk_expr:nNnN` to test or not whether the given expression is valid. The idea is to evaluate the expression within a brace group (to catch trailing `\use_none:nn` or similar), then test that the result is

what we expect. This is done by turning it to an integer and hitting that with `\tex_roman numeral:D` after replacing the first character by `-0`. If all goes well, that primitive finds a non-positive integer and gives an empty output. If the original expression evaluation stopped early it leaves a trailing `\tex_relax:D`, which stops the second evaluation (used to convert to integer) before it encounters the final `\tex_relax:D`. Since `\tex_roman numeral:D` does not absorb `\tex_relax:D` the output will be nonempty. Note that `#3` is empty except for mu expressions for which it is `\etex_mutoglu e:D` to avoid an “incompatible glue units” error. Note also that if we had omitted the first `\tex_relax:D` then for instance `1+2\relax+3` would incorrectly be accepted as a valid integer expression.

```

1676 \__kernel_if_debug:TF
1677 {
1678   \exp_args:Nc \cs_set_protected:Npn { __debug_check-expressions_on: }
1679   {
1680     \cs_set:Npn \__kernel_chk_expr:nNnN ##1##2
1681     {
1682       \__debug_suspended:T { ##1 \use_none:nnnnnn }
1683       \exp_after:wN \__debug_chk_expr_aux:nNnN
1684       \exp_after:wN { \tex_the:D ##2 ##1 \scan_stop: }
1685       ##2
1686     }
1687   }
1688   \exp_args:Nc \cs_set_protected:Npn { __debug_check-expressions_off: }
1689   { \cs_set:Npn \__kernel_chk_expr:nNnN ##1##2##3##4 {##1} }
1690   \use:c { __debug_check-expressions_off: }
1691   \cs_set:Npn \__debug_chk_expr_aux:nNnN #1#2#3#4
1692   {
1693     \tl_if_empty:oF
1694     {
1695       \tex_romannumeral:D - 0
1696       \exp_after:wN \use_none:n
1697       \int_value:w #3 #2 #1 \scan_stop:
1698     }
1699     {
1700       \__kernel_msg_expandable_error:nnnn
1701       { kernel } { expr } {#4} {#1}
1702     }
1703     #1
1704   }
1705 }
1706 { }
```

(End definition for `__debug_check-expressions_on:` and others.)

`__debug_log-functions_on:` These two functions (corresponding to the `expl3` option `log-functions`) control whether
`__debug_log-functions_off:` `__kernel_debug_log:x` writes to the log file or not. Since `\iow_log:x` does not yet
`__kernel_debug_log:x` have its final definition we do not use `\cs_set_eq:NN` (not defined yet anyway). Once
everything is defined, turn logging on or off depending on what option was given. When
debugging is not enabled, simply produce an error.

```

1707 \__kernel_if_debug:TF
1708 {
1709   \exp_args:Nc \cs_set_protected:Npn { __debug_log-functions_on: }
1710   {
```

```

1711         \cs_set_protected:Npn \__kernel_debug_log:x
1712         { \__debug_suspended:T \use_none:nn \iow_log:x }
1713     }
1714 \exp_args:Nc \cs_set_protected:Npn { __debug_log-functions_off: }
1715 { \cs_set_protected:Npn \__kernel_debug_log:x { \use_none:n } }
1716 \tex_ifodd:D \l@expl@log@functions@bool
1717 \use:c { __debug_log-functions_on: }
1718 \else:
1719 \use:c { __debug_log-functions_off: }
1720 \fi:
1721 }
1722 { }

```

(End definition for `__debug_log-functions_on:`, `__debug_log-functions_off:`, and `__kernel-debug_log:x`.)

`__debug_deprecation_on:` Some commands were more recently deprecated and not yet removed; only make these into errors if the user requests it. This relies on two token lists, mostly filled up by calls to `__kernel_patch_deprecation:nnNNpn` in each module.

```

\__debug_deprecation_off:
\__kernel_deprecation_code:nn
\g__debug_deprecation_on_tl
\g__debug_deprecation_off_tl
1723 \__kernel_if_debug:TF
1724 {
1725     \cs_set_protected:Npn \__debug_deprecation_on:
1726     { \g__debug_deprecation_on_tl }
1727     \cs_set_protected:Npn \__debug_deprecation_off:
1728     { \g__debug_deprecation_off_tl }
1729     \cs_set_nopar:Npn \g__debug_deprecation_on_tl { }
1730     \cs_set_nopar:Npn \g__debug_deprecation_off_tl { }
1731     \cs_set_protected:Npn \__kernel_deprecation_code:nn #1#2
1732     {
1733         \tl_gput_right:Nn \g__debug_deprecation_on_tl {#1}
1734         \tl_gput_right:Nn \g__debug_deprecation_off_tl {#2}
1735     }
1736 }
1737 {
1738     \cs_set_protected:Npn \__kernel_deprecation_code:nn #1#2 { }
1739 }

```

(End definition for `__debug_deprecation_on:` and others.)

`_kernel_patch_deprecation:nnNNpn` Grab a definition (at present, must be `\cs_new_protected:Npn` or `\cs_new:Npn`). Add to `\g__debug_deprecation_on_tl` some code that makes the defined macro #3 outer (and defines it as an error). Add to `\g__debug_deprecation_off_tl` the definition itself. In both cases we undefine the token with `\tex_let:D` to avoid taking a potentially outer macro as the argument of some `expl3` function. Finally, define the macro itself: if it is protected, make it produce a warning then redefine and call itself. The macro initially takes no parameters: together with the x-expanding assignment and `\exp_not:n` this gives a convenient way of storing the macro's definition in itself in order to only produce the warning once for each macro. If debugging is disabled, `__kernel_patch_deprecation:nnNNpn` lets the definition happen.

```

1740 \__kernel_if_debug:TF
1741 {
1742     \cs_set_protected:Npn \__kernel_patch_deprecation:nnNNpn #1#2#3#4#5#
1743     {
1744         \if_meaning:w \cs_new_protected:Npn #3

```

```

1745     \exp_after:wN \use_i:nn
1746   \else:
1747     \if_meaning:w \cs_new:Npn #3
1748       \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
1749     \else:
1750       \__kernel_msg_error:nnx { kernel } { debug-unpatchable }
1751       { \token_to_str:N #3 ~(for-deprecation) }
1752       \exp_after:wN \exp_after:wN \exp_after:wN \use_none:nn
1753     \fi:
1754   \fi:
1755   { \__debug_deprecation_aux:nnNnn {#1} {#2} #4 {#5} }
1756   { \__debug_deprecation_expandable:nnNnn {#1} {#2} #4 {#5} }
1757 }
1758 \cs_set_protected:Npn \__debug_deprecation_aux:nnNnn #1#2#3#4#5
1759 {
1760   \tl_gput_right:Nn \g__debug_deprecation_on_tl
1761   {
1762     \tex_let:D #3 \scan_stop:
1763     \__kernel_deprecation_error:Nnn #3 {#2} {#1}
1764   }
1765   \tl_gput_right:Nn \g__debug_deprecation_off_tl
1766   {
1767     \tex_let:D #3 \scan_stop:
1768     \cs_set_protected:Npn #3 #4 {#5}
1769   }
1770   \cs_new_protected:Npx #3
1771   {
1772     \exp_not:N \__kernel_msg_warning:nnxxx
1773     { kernel } { deprecated-command }
1774     {#1} { \token_to_str:N #3 } { \tl_to_str:n {#2} }
1775     \exp_not:n { \cs_gset_protected:Npn #3 #4 {#5} }
1776     \exp_not:N #3
1777   }
1778 }
1779 \cs_set_protected:Npn \__debug_deprecation_expandable:nnNnn #1#2#3#4#5
1780 {
1781   \tl_gput_right:Nn \g__debug_deprecation_on_tl
1782   {
1783     \tex_let:D #3 \scan_stop:
1784     \__kernel_deprecation_error:Nnn #3 {#2} {#1}
1785   }
1786   \tl_gput_right:Nn \g__debug_deprecation_off_tl
1787   {
1788     \tex_let:D #3 \scan_stop:
1789     \cs_set:Npn #3 #4 {#5}
1790   }
1791   \cs_new:Npn #3 #4 {#5}
1792 }
1793 }
1794 { \cs_set_protected:Npn \__kernel_patch_deprecation:nnNNpn #1#2 { } }

```

(End definition for __kernel_patch_deprecation:nnNNpn and __debug_deprecation_aux:nnNnn.)

__kernel_patch:nnNNpn When debugging is not enabled, __kernel_patch:nnNNpn and __kernel_patch_-\nconditional:nnNNpnn throw the patch away. Otherwise they can be followed by \cs_-\n__debug_patch_aux:nnnn\n__debug_patch_auxii:nnnn

`new:Npn` (or similar), and `\prg_new_conditional:Npnn` (or similar), respectively. In each case, grab the name of the function to be defined and its parameters then insert tokens before and/or after the definition.

```

1795 \__kernel_if_debug:TF
1796 {
1797   \cs_set_protected:Npn \__kernel_patch:nnNNpn #1#2#3#4#5#
1798     { \__debug_patch_aux:nnnn {#1} {#2} { #3 #4 #5 } }
1799   \cs_set_protected:Npn \__kernel_patch_conditional:nnNpnn #1#2#3#4#
1800     { \__debug_patch_auxii:nnnn {#1} { #2 #3 #4 } }
1801   \cs_set_protected:Npn \__debug_patch_aux:nnnn #1#2#3#4
1802     { #3 { #1 #4 #2 } }
1803   \cs_set_protected:Npn \__debug_patch_auxii:nnnn #1#2#3#4
1804     { #2 {#3} { #1 #4 } }
1805 }
1806 {
1807   \cs_set_protected:Npn \__kernel_patch:nnNNpn #1#2 { }
1808   \cs_set_protected:Npn \__kernel_patch_conditional:nnNpnn #1 { }
1809 }

```

(End definition for `__kernel_patch:nnNNpn` and others.)

<pre> __kernel_patch_args:nnNpn __kernel_patch_conditional_args:nnNpnn __kernel_patch_args:nnnNNpn __kernel_patch_conditional_args:nnnNNpnn __debug_tmp:w __debug_patch_args_aux:nnnNNnn __debug_patch_args_aux:nnnNNnnn __debug_patch_args_aux:nnnn </pre>	<p>See <code>__kernel_patch:nnNNpn</code>. The first argument is something like <code>{#1}{(#2)}</code>. Define a temporary macro using the <i>parameters</i> and <i>code</i> of the definition that follows, then expand that temporary macro in front of the first argument to obtain new <i>code</i>. Then perform the definition as if that new <i>code</i> was directly typed in the file.</p> <pre> 1810 \cs_set_protected:Npn __kernel_patch_args:nnNpn 1811 { __kernel_patch_args:nnnNNpn { } { } } 1812 \cs_set_protected:Npn __kernel_patch_conditional_args:nnNpnn 1813 { __kernel_patch_conditional_args:nnnNNpnn { } { } } 1814 __kernel_if_debug:TF 1815 { 1816 \cs_set_protected:Npn __kernel_patch_args:nnnNNpn #1#2#3#4#5#6# 1817 { __debug_patch_args_aux:nnnNNnn {#1} {#2} {#3} #4 #5 {#6} } 1818 \cs_set_protected:Npn __kernel_patch_conditional_args:nnnNNpnn 1819 #1#2#3#4#5#6# 1820 { __debug_patch_args_aux:nnnNNnn {#1} {#2} {#3} #4 #5 {#6} } 1821 \cs_set_protected:Npn __debug_patch_args_aux:nnnNNnn #1#2#3#4#5#6#7 1822 { 1823 \cs_set:Npn __debug_tmp:w #6 {#7} 1824 \exp_after:wN __debug_patch_args_aux:nnnn \exp_after:wN 1825 { __debug_tmp:w #3 } { #4 #5 #6 } {#1} {#2} 1826 } 1827 \cs_set_protected:Npn __debug_patch_args_aux:nnnNNnnn #1#2#3#4#5#6#7#8 1828 { 1829 \cs_set:Npn __debug_tmp:w #6 {#8} 1830 \exp_after:wN __debug_patch_args_aux:nnnn \exp_after:wN 1831 { __debug_tmp:w #3 } { #4 #5 #6 {#7} } {#1} {#2} 1832 } 1833 \cs_set_protected:Npn __debug_patch_args_aux:nnnn #1#2#3#4 1834 { #2 { #3 #1 #4 } } 1835 } 1836 { 1837 \cs_set_protected:Npn __kernel_patch_args:nnnNNpn #1#2#3 { } 1838 \cs_set_protected:Npn __kernel_patch_conditional_args:nnnNNpnn </pre>
---	---

```

1839      #1#2#3 { }
1840    }

```

(End definition for `_kernel_patch_args:nNpn` and others.)

4.7 Conditional processing and definitions

```

1841 <@@=prg>

```

Underneath any predicate function (`_p`) or other conditional forms (TF, etc.) is a built-in logic saying that it after all of the testing and processing must return the *<state>* this leaves T_EX in. Therefore, a simple user interface could be something like

```

\if_meaning:w #1#2
\prg_return_true:
\else:
\if_meaning:w #1#3
\prg_return_true:
\else:
\prg_return_false:
\fi:
\fi:

```

Usually, a T_EX programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the T_EX programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:` The idea here is that `\exp:w` expands fully any `\else:` and `\fi:` that are waiting to be discarded, before reaching the `\exp_end:` which leaves an empty expansion. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

```

1842 \cs_set:Npn \prg_return_true:
1843 { \exp_after:wN \use_i:nn \exp:w }
1844 \cs_set:Npn \prg_return_false:
1845 { \exp_after:wN \use_ii:nn \exp:w }

```

An extended state space could be implemented by including a more elaborate function in place of `\use_i:nn/\use_ii:nn`. Provided two arguments are absorbed then the code would work.

(End definition for `\prg_return_true:` and `\prg_return_false:`. These functions are documented on page 94.)

`\prg_set_conditional:Npnn` The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those `Npnn` type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed *<{<name>}<{<signature>}<{<boolean>}<{<set or new>}<{<maybe protected>}<{<parameters>}<{TF,...}<{<code>}<* to the auxiliary function responsible for defining all conditionals.

```

1846 \cs_set_protected:Npn \prg_set_conditional:Npnn
1847 { \_prg_generate_conditional_parm:nnNpnn { set } { } }
1848 \cs_set_protected:Npn \prg_new_conditional:Npnn
1849 { \_prg_generate_conditional_parm:nnNpnn { new } { } }

```

```

1850 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn
1851   { \__prg_generate_conditional_parm:nnNpnn { set } { _protected } }
1852 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn
1853   { \__prg_generate_conditional_parm:nnNpnn { new } { _protected } }
1854 \cs_set_protected:Npn \__prg_generate_conditional_parm:nnNpnn #1#2#3#4#
1855   {
1856     \use:x
1857     {
1858       \__prg_generate_conditional:nnNnnnnn
1859       \cs_split_function:N #3
1860     }
1861     {#1} {#2} {#4}
1862   }

```

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 92.)

```

\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\__prg_generate_conditional_count:nnNnn
\__prg_generate_conditional_count:nnNnnnn

```

The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary generates the parameter text from the number of letters in the signature. Then feed `{<name>}{<signature>}{<boolean>}{<set or new>}{<maybe protected>}{<parameters>}{TF,...}{<code>}` to the auxiliary function responsible for defining all conditionals. If the `<signature>` has more than 9 letters, the definition is aborted since T_EX macros have at most 9 arguments. The erroneous case where the function name contains no colon is captured later.

```

1863 \cs_set_protected:Npn \prg_set_conditional:Nnn
1864   { \__prg_generate_conditional_count:nnNnn { set } { } }
1865 \cs_set_protected:Npn \prg_new_conditional:Nnn
1866   { \__prg_generate_conditional_count:nnNnn { new } { } }
1867 \cs_set_protected:Npn \prg_set_protected_conditional:Nnn
1868   { \__prg_generate_conditional_count:nnNnn { set } { _protected } }
1869 \cs_set_protected:Npn \prg_new_protected_conditional:Nnn
1870   { \__prg_generate_conditional_count:nnNnn { new } { _protected } }
1871 \cs_set_protected:Npn \__prg_generate_conditional_count:nnNnn #1#2#3
1872   {
1873     \use:x
1874     {
1875       \__prg_generate_conditional_count:nnNnnnn
1876       \cs_split_function:N #3
1877     }
1878     {#1} {#2}
1879   }
1880 \cs_set_protected:Npn \__prg_generate_conditional_count:nnNnnnn #1#2#3#4#5
1881   {
1882     \__kernel_cs_parm_from_arg_count:nnF
1883     { \__prg_generate_conditional:nnNnnnnn {#1} {#2} #3 {#4} {#5} }
1884     { \tl_count:n {#2} }
1885     {
1886       \__kernel_msg_error:nnxx { kernel } { bad-number-of-arguments }
1887       { \token_to_str:c { #1 : #2 } }
1888       { \tl_count:n {#2} }
1889       \use_none:nn
1890     }

```

```
1891 }
```

(End definition for `\prg_set_conditional:Nnn` and others. These functions are documented on page 92.)

```
\__prg_generate_conditional:nnNnnnnn
\__prg_generate_conditional:nnnnnnw
```

The workhorse here is going through a list of desired forms, *i.e.*, p, TF, T and F. The first three arguments come from splitting up the base form of the conditional, which gives the name, signature and a boolean to signal whether or not there was a colon in the name. In the absence of a colon, we throw an error and don't define any conditional. The fourth and fifth arguments build up the defining function. The sixth is the parameters to use (possibly empty), the seventh is the list of forms to define, the eighth is the replacement text which we will augment when defining the forms. The use of `\tl_to_str:n` makes the later loop more robust.

```
1892 \cs_set_protected:Npn \__prg_generate_conditional:nnNnnnnn #1#2#3#4#5#6#7#8
1893 {
1894   \if_meaning:w \c_false_bool #3
1895     \__kernel_msg_error:nnx { kernel } { missing-colon }
1896     { \token_to_str:c {#1} }
1897     \exp_after:wN \use_none:nn
1898   \fi:
1899   \use:x
1900   {
1901     \exp_not:N \__prg_generate_conditional:nnnnnnw
1902     \exp_not:n { {#4} {#5} {#1} {#2} {#6} {#8} }
1903     \tl_to_str:n {#7}
1904     \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
1905   }
1906 }
```

Looping through the list of desired forms. First are six arguments and seventh is the form. Use the form to call the correct type. If the form does not exist, the `\use:c` construction results in `\relax`, and the error message is displayed (unless the form is empty, to allow for {T, , F}), then `\use_none:nnnnnnn` cleans up. Otherwise, the error message is removed by the variant form.

```
1907 \cs_set_protected:Npn \__prg_generate_conditional:nnnnnnw #1#2#3#4#5#6#7 ,
1908 {
1909   \if_meaning:w \q_recursion_tail #7
1910     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1911   \fi:
1912   \use:c { __prg_generate_ #7 _form:wnnnnnnn }
1913   \tl_if_empty:nF {#7}
1914   {
1915     \__kernel_msg_error:nnxx
1916     { kernel } { conditional-form-unknown }
1917     {#7} { \token_to_str:c { #3 : #4 } }
1918   }
1919   \use_none:nnnnnnn
1920   \q_stop
1921   {#1} {#2} {#3} {#4} {#5} {#6}
1922   \__prg_generate_conditional:nnnnnnw {#1} {#2} {#3} {#4} {#5} {#6}
1923 }
```

(End definition for `__prg_generate_conditional:nnNnnnnn` and `__prg_generate_conditional:nnnnnnw`.)

```

\__prg_generate_p_form:wnnnnnnn
\__prg_generate_TF_form:wnnnnnnn
\__prg_generate_T_form:wnnnnnnn
\__prg_generate_F_form:wnnnnnnn

```

How to generate the various forms. Those functions take the following arguments: 1: **set** or **new**, 2: empty or **_protected**, 3: function name 4: signature, 5: parameter text (or empty), 6: replacement. Remember that the logic-returning functions expect two arguments to be present after **\exp_end::** notice the construction of the different variants relies on this, and that the TF and F variants will be slightly faster than the T version. The **p** form is only valid for expandable tests, we check for that by making sure that the second argument is empty.

```

1924 \cs_set_protected:Npn \__prg_generate_p_form:wnnnnnnn
1925   #1 \q_stop #2#3#4#5#6#7
1926   {
1927     \if_meaning:w \scan_stop: #3 \scan_stop:
1928       \exp_after:wN \use_i:nn
1929     \else:
1930       \exp_after:wN \use_ii:nn
1931     \fi:
1932     {
1933       \exp_args:cc { cs_ #2 #3 :Npn } { #4 _p: #5 } #6
1934       { #7 \exp_end: \c_true_bool \c_false_bool }
1935     }
1936     {
1937       \__kernel_msg_error:nxx { kernel } { protected-predicate }
1938       { \token_to_str:c { #4 _p: #5 } }
1939     }
1940   }
1941 \cs_set_protected:Npn \__prg_generate_T_form:wnnnnnnn
1942   #1 \q_stop #2#3#4#5#6#7
1943   {
1944     \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 T } #6
1945     { #7 \exp_end: \use:n \use_none:n }
1946   }
1947 \cs_set_protected:Npn \__prg_generate_F_form:wnnnnnnn
1948   #1 \q_stop #2#3#4#5#6#7
1949   {
1950     \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 F } #6
1951     { #7 \exp_end: { } }
1952   }
1953 \cs_set_protected:Npn \__prg_generate_TF_form:wnnnnnnn
1954   #1 \q_stop #2#3#4#5#6#7
1955   {
1956     \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 TF } #6
1957     { #7 \exp_end: }
1958   }

```

(End definition for `__prg_generate_p_form:wnnnnnnn` and others.)

```

\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn
\__prg_set_eq_conditional:NNNn

```

The setting-equal functions. Split both functions and feed $\{\langle name_1 \rangle\}$ $\{\langle signature_1 \rangle\}$ $\langle boolean_1 \rangle$ $\{\langle name_2 \rangle\}$ $\{\langle signature_2 \rangle\}$ $\langle boolean_2 \rangle$ $\langle copying\ function \rangle$ $\langle conditions \rangle$, **\q-recursion_tail**, **\q-recursion_stop** to a first auxiliary.

```

1959 \cs_set_protected:Npn \prg_set_eq_conditional:NNn
1960   { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
1961 \cs_set_protected:Npn \prg_new_eq_conditional:NNn
1962   { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
1963 \cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4

```

```

1964 {
1965   \use:x
1966   {
1967     \exp_not:N \__prg_set_eq_conditional:nnNnnNNw
1968     \cs_split_function:N #2
1969     \cs_split_function:N #3
1970     \exp_not:N #1
1971     \tl_to_str:n {#4}
1972     \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
1973   }
1974 }

```

(End definition for \prg_set_eq_conditional:NNn, \prg_new_eq_conditional:NNn, and __prg_set_eq_conditional:NNNn. These functions are documented on page 93.)

__prg_set_eq_conditional:nnNnnNNw Split the function to be defined, and setup a manual clist loop over argument #6 of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don't know how to build conditionals, hence abort. Call the looping macro, with arguments {<name₁>} {<signature₁>} {<name₂>} {<signature₂>} {<copying function>} and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```

1975 \cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
1976 {
1977   \if_meaning:w \c_false_bool #3
1978   \__kernel_msg_error:nxx { kernel } { missing-colon }
1979   { \token_to_str:c {#1} }
1980   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1981   \fi:
1982   \if_meaning:w \c_false_bool #6
1983   \__kernel_msg_error:nxx { kernel } { missing-colon }
1984   { \token_to_str:c {#4} }
1985   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1986   \fi:
1987   \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
1988 }
1989 \cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
1990 {
1991   \if_meaning:w \q_recursion_tail #6
1992   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1993   \fi:
1994   \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
1995   \tl_if_empty:nF {#6}
1996   {
1997     \__kernel_msg_error:nxxx
1998     { kernel } { conditional-form-unknown }
1999     {#6} { \token_to_str:c { #1 : #2 } }
2000   }
2001   \use_none:nnnnnn
2002   \q_stop
2003   #5 {#1} {#2} {#3} {#4}
2004   \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
2005 }
2006 \__kernel_patch:nnNNpn

```

```

2007 { \_kernel_chk_cs_exist:c { #5 _p : #6 } } { }
2008 \cs_set:Npn \_prg_set_eq_conditional_p_form:wNnnnn #1 \q_stop #2#3#4#5#6
2009 { #2 { #3 _p : #4 } { #5 _p : #6 } }
2010 \_kernel_patch:nnNNpn
2011 { \_kernel_chk_cs_exist:c { #5 : #6 TF } } { }
2012 \cs_set:Npn \_prg_set_eq_conditional_TF_form:wNnnnn #1 \q_stop #2#3#4#5#6
2013 { #2 { #3 : #4 TF } { #5 : #6 TF } }
2014 \_kernel_patch:nnNNpn
2015 { \_kernel_chk_cs_exist:c { #5 : #6 T } } { }
2016 \cs_set:Npn \_prg_set_eq_conditional_T_form:wNnnnn #1 \q_stop #2#3#4#5#6
2017 { #2 { #3 : #4 T } { #5 : #6 T } }
2018 \_kernel_patch:nnNNpn
2019 { \_kernel_chk_cs_exist:c { #5 : #6 F } } { }
2020 \cs_set:Npn \_prg_set_eq_conditional_F_form:wNnnnn #1 \q_stop #2#3#4#5#6
2021 { #2 { #3 : #4 F } { #5 : #6 F } }

```

(End definition for `_prg_set_eq_conditional:nnNnnNNw` and others.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using `\if:w` but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

`\c_true_bool` Here are the canonical boolean values.

`\c_false_bool`

```

2022 \tex_chardef:D \c_true_bool = 1 ~
2023 \tex_chardef:D \c_false_bool = 0 ~

```

(End definition for `\c_true_bool` and `\c_false_bool`. These variables are documented on page 19.)

4.8 Dissecting a control sequence

2024 `<@@=cs>`

`_cs_count_signature:N` `_cs_count_signature:N <function>`

Splits the `<function>` into the `<name>` (i.e. the part before the colon) and the `<signature>` (i.e. after the colon). The `<number>` of tokens in the `<signature>` is then left in the input stream. If there was no `<signature>` then the result is the marker value `-1`.

`_cs_get_function_name:N` ★ `_cs_get_function_name:N <function>`

Splits the `<function>` into the `<name>` (i.e. the part before the colon) and the `<signature>` (i.e. after the colon). The `<name>` is then left in the input stream without the escape character present made up of tokens with category code 12 (other).

`_cs_get_function_signature:N` ★ `_cs_get_function_signature:N <function>`

Splits the `<function>` into the `<name>` (i.e. the part before the colon) and the `<signature>` (i.e. after the colon). The `<signature>` is then left in the input stream made up of tokens with category code 12 (other).

<code>__cs_tmp:w</code>	Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).
<code>\cs_to_str:N</code>	This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there are different cases:
<code>__cs_to_str:N</code>	
<code>__cs_to_str:w</code>	

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of the `\escapechar` is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N _` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TeX reads `__cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `\tex_romannumeral:D`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a the space from `\token_to_str:N _`, and the auxiliary `__cs_to_str:w` is expanded, feeding – as a second character for the test; the test is false, and TeX skips to `\fi:`, then performs `\token_to_str:N`, and stops the `\tex_romannumeral:D` with `\c_zero`. The last case is that the escape character is itself a space. In this case, the `\if:w` test is true, and the auxiliary `__cs_to_str:w` comes into play, inserting `-\int_value:w`, which expands `\c_zero` to the character 0. The initial `\tex_romannumeral:D` then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the expansion of `\tex_romannumeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```

2025 \cs_set:Npn \cs_to_str:N
2026   {

```

We implement the expansion scheme using `\tex_romannumeral:D` terminating it with `\c_zero` rather than using `\exp:w` and `\exp_end:` as we normally do. The reason is that the code heavily depends on terminating the expansion with `\c_zero` so we make this dependency explicit.

```

2027   \tex_romannumeral:D
2028   \if:w \token_to_str:N \ \__cs_to_str:w \fi:
2029   \exp_after:wN \__cs_to_str:N \token_to_str:N
2030 }
2031 \cs_set:Npn \__cs_to_str:N #1 { \c_zero }
2032 \cs_set:Npn \__cs_to_str:w #1 \__cs_to_str:N
2033 { - \int_value:w \fi: \exp_after:wN \c_zero }

```

If speed is a concern we could use `\csstring` in LuaTeX. For the empty csname that primitive gives an empty result while the current `\cs_to_str:N` gives incorrect results in all engines (this is impossible to fix without huge performance hit).

(End definition for `\cs_to_str:N`, `__cs_to_str:N`, and `__cs_to_str:w`. This function is documented on page 16.)

`\cs_split_function:N`

This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean $\langle true \rangle$ or $\langle false \rangle$ is returned with $\langle true \rangle$ for when there is a colon in the function and $\langle false \rangle$ if there is not.

We cannot use `:` directly as it has the wrong category code so an `x`-type expansion is used to force the conversion.

First ensure that we actually get a properly evaluated string by expanding `\cs_to_str:N` twice. If the function contained a colon, the auxiliary takes as `#1` the function name, delimited by the first colon, then the signature `#2`, delimited by `\q_mark`, then `\c_true_bool` as `#3`, and `#4` cleans up until `\q_stop`. Otherwise, the `#1` contains the function name and `\q_mark \c_true_bool`, `#2` is empty, `#3` is `\c_false_bool`, and `#4` cleans up. The second auxiliary trims the trailing `\q_mark` from the function name if present (that is, if the original function had no colon).

```

2034 \cs_set_protected:Npn \__cs_tmp:w #1
2035 {
2036   \cs_set:Npn \cs_split_function:N ##1
2037   {
2038     \exp_after:wN \exp_after:wN \exp_after:wN
2039     \__cs_split_function_auxi:w
2040     \cs_to_str:N ##1 \q_mark \c_true_bool
2041     #1 \q_mark \c_false_bool \q_stop
2042   }
2043   \cs_set:Npn \__cs_split_function_auxi:w
2044     ##1 #1 ##2 \q_mark ##3##4 \q_stop
2045     { \__cs_split_function_auxii:w ##1 \q_mark \q_stop {##2} ##3 }
2046   \cs_set:Npn \__cs_split_function_auxii:w ##1 \q_mark ##2 \q_stop
2047     { {##1} }
2048 }
2049 \exp_after:wN \__cs_tmp:w \token_to_str:N :
```

(End definition for `\cs_split_function:N`, `__cs_split_function_auxi:w`, and `__cs_split_function_auxii:w`. This function is documented on page 16.)

4.9 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive `\relax` token. A control sequence is said to be *free* (to be defined) if it does not already exist.

`\cs_if_exist_p:N`

`\cs_if_exist_p:c`

`\cs_if_exist:NTF`

`\cs_if_exist:cTF`

Two versions for checking existence. For the `N` form we firstly check for `\scan_stop:` and then if it is in the hash table. There is no problem when inputting something like `\else:` or `\fi:` as \TeX will only ever skip input in case the token tested against is `\scan_stop:`.

```

2050 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
2051 {
2052   \if_meaning:w #1 \scan_stop:
2053   \prg_return_false:
2054   \else:
2055     \if_cs_exist:N #1
2056     \prg_return_true:
2057   \else:
```

```

2058     \prg_return_false:
2059     \fi:
2060     \fi:
2061 }

```

For the `c` form we firstly check if it is in the hash table and then for `\scan_stop`: so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```

2062 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
2063 {
2064     \if_cs_exist:w #1 \cs_end:
2065     \exp_after:wN \use_i:nn
2066     \else:
2067     \exp_after:wN \use_ii:nn
2068     \fi:
2069     {
2070     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
2071     \prg_return_false:
2072     \else:
2073     \prg_return_true:
2074     \fi:
2075     }
2076     \prg_return_false:
2077 }

```

(End definition for `\cs_if_exist:NTF`. This function is documented on page 20.)

`\cs_if_free_p:N`

The logical reversal of the above.

`\cs_if_free_p:c`

`\cs_if_free:NTF`

`\cs_if_free:cTF`

```

2078 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
2079 {
2080     \if_meaning:w #1 \scan_stop:
2081     \prg_return_true:
2082     \else:
2083     \if_cs_exist:N #1
2084     \prg_return_false:
2085     \else:
2086     \prg_return_true:
2087     \fi:
2088     \fi:
2089 }
2090 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
2091 {
2092     \if_cs_exist:w #1 \cs_end:
2093     \exp_after:wN \use_i:nn
2094     \else:
2095     \exp_after:wN \use_ii:nn
2096     \fi:
2097     {
2098     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
2099     \prg_return_true:
2100     \else:
2101     \prg_return_false:

```

```

2102     \fi:
2103   }
2104   { \prg_return_true: }
2105 }

```

(End definition for `\cs_if_free:NTF`. This function is documented on page 20.)

`\cs_if_exist_use:N` The `\cs_if_exist_use:...` functions cannot be implemented as conditionals because the true branch must leave both the control sequence itself and the true code in the input stream. For the `c` variants, we are careful not to put the control sequence in the hash table if it does not exist. In LuaTeX we could use the `\lastnamedcs` primitive.

```

2106 \cs_set:Npn \cs_if_exist_use:NTF #1#2
2107 { \cs_if_exist:NTF #1 { #1 #2 } }
2108 \cs_set:Npn \cs_if_exist_use:NF #1
2109 { \cs_if_exist:NTF #1 { #1 } }
2110 \cs_set:Npn \cs_if_exist_use:NT #1 #2
2111 { \cs_if_exist:NTF #1 { #1 #2 } { } }
2112 \cs_set:Npn \cs_if_exist_use:N #1
2113 { \cs_if_exist:NTF #1 { #1 } { } }
2114 \cs_set:Npn \cs_if_exist_use:cTF #1#2
2115 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
2116 \cs_set:Npn \cs_if_exist_use:cF #1
2117 { \cs_if_exist:cTF {#1} { \use:c {#1} } }
2118 \cs_set:Npn \cs_if_exist_use:cT #1#2
2119 { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
2120 \cs_set:Npn \cs_if_exist_use:c #1
2121 { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }

```

(End definition for `\cs_if_exist_use:NTF`. This function is documented on page 15.)

4.10 Preliminaries for new functions

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called `\..._new`. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The next few definitions here are only temporary, they will be redefined later on.

`__kernel_msg_error:nxxx` If an internal error occurs before L^AT_EX₃ has loaded `l3msg` then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response. Setting the `\newlinechar` is needed, to turn `^^J` into a proper line break in plain T_EX.

```

2122 \cs_set_protected:Npn \__kernel_msg_error:nxxx #1#2#3#4
2123 {
2124   \tex_newlinechar:D = '\^^J \scan_stop:
2125   \tex_errmessage:D
2126   {
2127     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
2128     Argh,~internal~LaTeX3~error! ^^J ^^J
2129     Module ~ #1 , ~ message~name~"#2": ^^J
2130     Arguments~'#3'~and~'#4' ^^J ^^J

```

```

2131         This~is~one~for~The~LaTeX3~Project:~bailing~out
2132     }
2133     \tex_end:D
2134 }
2135 \cs_set_protected:Npn \__kernel_msg_error:nnx #1#2#3
2136 { \__kernel_msg_error:nnxx {#1} {#2} {#3} { } }
2137 \cs_set_protected:Npn \__kernel_msg_error:nn #1#2
2138 { \__kernel_msg_error:nnxx {#1} {#2} { } { } }

```

(End definition for __kernel_msg_error:nnxx, __kernel_msg_error:nnx, and __kernel_msg_error:nn.)

\msg_line_context: Another one from l3msg which will be altered later.

```

2139 \cs_set:Npn \msg_line_context:
2140 { on~line~ \tex_the:D \tex_inputlineno:D }

```

(End definition for \msg_line_context:. This function is documented on page 135.)

\iow_log:x We define a routine to write only to the log file. And a similar one for writing to both
\iow_term:x the log file and the terminal. These will be redefined later by l3io.

```

2141 \cs_set_protected:Npn \iow_log:x
2142 { \tex_immediate:D \tex_write:D -1 }
2143 \cs_set_protected:Npn \iow_term:x
2144 { \tex_immediate:D \tex_write:D 16 }

```

(End definition for \iow_log:n. This function is documented on page 145.)

__kernel_chk_if_free_cs:N This command is called by \cs_new_nopar:Npn and \cs_new_eq:NN etc. to make sure
__kernel_chk_if_free_cs:c that the argument sequence is not already in use. If it is, an error is signalled. It checks
if $\langle csname \rangle$ is undefined or \scan_stop:. Otherwise an error message is issued. We have
to make sure we don't put the argument into the conditional processing since it may be
an \if... type function!

```

2145 \__kernel_patch:nnNpn { }
2146 {
2147     \__kernel_debug_log:x
2148     { Defining~\token_to_str:N #1~ \msg_line_context: }
2149 }
2150 \cs_set_protected:Npn \__kernel_chk_if_free_cs:N #1
2151 {
2152     \cs_if_free:NF #1
2153     {
2154         \__kernel_msg_error:nnxx { kernel } { command-already-defined }
2155         { \token_to_str:N #1 } { \token_to_meaning:N #1 }
2156     }
2157 }
2158 \cs_set_protected:Npn \__kernel_chk_if_free_cs:c
2159 { \exp_args:Nc \__kernel_chk_if_free_cs:N }

```

(End definition for __kernel_chk_if_free_cs:N.)

4.11 Defining new functions

2160 <@@=cs>

`\cs_new_nopar:Npn` Function which check that the control sequence is free before defining it.

```

2161 \cs_set:Npn \__cs_tmp:w #1#2
2162 {
2163   \cs_set_protected:Npn #1 ##1
2164   {
2165     \__kernel_chk_if_free_cs:N ##1
2166     #2 ##1
2167   }
2168 }
2169 \__cs_tmp:w \cs_new_nopar:Npn \cs_gset_nopar:Npn
2170 \__cs_tmp:w \cs_new_nopar:Npx \cs_gset_nopar:Npx
2171 \__cs_tmp:w \cs_new:Npn \cs_gset:Npn
2172 \__cs_tmp:w \cs_new:Npx \cs_gset:Npx
2173 \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
2174 \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
2175 \__cs_tmp:w \cs_new_protected:Npn \cs_gset_protected:Npn
2176 \__cs_tmp:w \cs_new_protected:Npx \cs_gset_protected:Npx

```

(End definition for `\cs_new_nopar:Npn` and others. These functions are documented on page 10.)

`\cs_set_nopar:cpn` Like `\cs_set_nopar:Npn` and `\cs_new_nopar:Npn`, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the `c` stands for `csname` argument, see the expansion module). Global versions are also provided.

`\cs_set_nopar:cpn` `\cs_set_nopar:cpn`(*string*)(*rep-text*) turns *string* into a `csname` and then assigns *rep-text* to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

2177 \cs_set:Npn \__cs_tmp:w #1#2
2178 { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
2179 \__cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
2180 \__cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
2181 \__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
2182 \__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
2183 \__cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
2184 \__cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End definition for `\cs_set_nopar:Npn`. This function is documented on page 10.)

`\cs_set:cpn` Variants of the `\cs_set:Npn` versions which make a `csname` out of the first arguments.
`\cs_set:cpx` We may also do this globally.

```

2185 \__cs_tmp:w \cs_set:cpn \cs_set:Npn
2186 \__cs_tmp:w \cs_set:cpx \cs_set:Npx
2187 \__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
2188 \__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
2189 \__cs_tmp:w \cs_new:cpn \cs_new:Npn
2190 \__cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End definition for `\cs_set:Npn`. This function is documented on page 10.)

<code>\cs_set_protected_nopar:cpn</code>	2191	<code>__cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn</code>	Variants of the <code>\cs_set_protected_nopar:Npn</code> versions which make a csname out of the first arguments. We may also do this globally.
<code>\cs_set_protected_nopar:cpx</code>	2192	<code>__cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx</code>	
<code>\cs_gset_protected_nopar:cpn</code>	2193	<code>__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn</code>	
<code>\cs_gset_protected_nopar:cpx</code>	2194	<code>__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx</code>	
<code>\cs_new_protected_nopar:cpn</code>	2195	<code>__cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn</code>	
<code>\cs_new_protected_nopar:cpx</code>	2196	<code>__cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx</code>	

(End definition for `\cs_set_protected_nopar:Npn`. This function is documented on page 11.)

<code>\cs_set_protected:cpn</code>	2197	<code>__cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn</code>	Variants of the <code>\cs_set_protected:Npn</code> versions which make a csname out of the first arguments. We may also do this globally.
<code>\cs_set_protected:cpx</code>	2198	<code>__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx</code>	
<code>\cs_gset_protected:cpn</code>	2199	<code>__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn</code>	
<code>\cs_gset_protected:cpx</code>	2200	<code>__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx</code>	
<code>\cs_new_protected:cpn</code>	2201	<code>__cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn</code>	
<code>\cs_new_protected:cpx</code>	2202	<code>__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx</code>	

(End definition for `\cs_set_protected:Npn`. This function is documented on page 10.)

4.12 Copying definitions

<code>\cs_set_eq:NN</code>	These macros allow us to copy the definition of a control sequence to another control sequence.
<code>\cs_set_eq:cN</code>	The = sign allows us to define funny char tokens like = itself or <code>_</code> with this function.
<code>\cs_set_eq:Nc</code>	For the definition of <code>\c_space_char{~}</code> to work we need the ~ after the =.
<code>\cs_set_eq:cc</code>	
<code>\cs_gset_eq:NN</code>	<code>\cs_set_eq:NN</code> is long to avoid problems with a literal argument of <code>\par</code> . While
<code>\cs_gset_eq:cN</code>	<code>\cs_new_eq:NN</code> will probably never be correct with a first argument of <code>\par</code> , define it
<code>\cs_gset_eq:Nc</code>	long in order to throw an “already defined” error rather than “runaway argument”.
<code>\cs_gset_eq:cc</code>	2203 <code>\cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }</code>
<code>\cs_new_eq:NN</code>	2204 <code>\cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }</code>
<code>\cs_new_eq:cN</code>	2205 <code>\cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }</code>
<code>\cs_new_eq:Nc</code>	2206 <code>\cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }</code>
<code>\cs_new_eq:cc</code>	2207 <code>\cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }</code>
	2208 <code>\cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }</code>
	2209 <code>\cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }</code>
	2210 <code>\cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }</code>
	2211 <code>\cs_new_protected:Npn \cs_new_eq:NN #1</code>
	2212 <code>{</code>
	2213 <code> _kernel_chk_if_free_cs:N #1</code>
	2214 <code> \tex_global:D \cs_set_eq:NN #1</code>
	2215 <code>}</code>
	2216 <code>\cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }</code>
	2217 <code>\cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }</code>
	2218 <code>\cs_new_protected:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }</code>

(End definition for `\cs_set_eq:NN`, `\cs_gset_eq:NN`, and `\cs_new_eq:NN`. These functions are documented on page 14.)

4.13 Undefining functions

`\cs_undefine:N` The following function is used to free the main memory from the definition of some function that isn't in use any longer. The `c` variant is careful not to add the control sequence to the hash table if it isn't there yet, and it also avoids nesting \TeX conditionals in case `#1` is unbalanced in this matter.

```

2219 \cs_new_protected:Npn \cs_undefine:N #1
2220 { \cs_gset_eq:NN #1 \tex_undefined:D }
2221 \cs_new_protected:Npn \cs_undefine:c #1
2222 {
2223   \if_cs_exist:w #1 \cs_end:
2224     \exp_after:wN \use:n
2225   \else:
2226     \exp_after:wN \use_none:n
2227   \fi:
2228   { \cs_gset_eq:cN {#1} \tex_undefined:D }
2229 }
```

(End definition for `\cs_undefine:N`. This function is documented on page 14.)

4.14 Generating parameter text from argument count

```

2230 <@@=cs>
```

`_kernel_cs_parm_from_arg_count:nnF` \LaTeX 3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text `{#1...#n}`, where n is the result of evaluating the second argument (as described in `\int_eval:n`). If the second argument gives a result outside the range $[0, 9]$, the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```

2231 \cs_set_protected:Npn \_kernel_cs_parm_from_arg_count:nnF #1#2
2232 {
2233   \exp_args:Nx \_cs_parm_from_arg_count_test:nnF
2234   {
2235     \exp_after:wN \exp_not:n
2236     \if_case:w \int_eval:n {#2}
2237       { }
2238     \or: { ##1 }
2239     \or: { ##1##2 }
2240     \or: { ##1##2##3 }
2241     \or: { ##1##2##3##4 }
2242     \or: { ##1##2##3##4##5 }
2243     \or: { ##1##2##3##4##5##6 }
2244     \or: { ##1##2##3##4##5##6##7 }
2245     \or: { ##1##2##3##4##5##6##7##8 }
2246     \or: { ##1##2##3##4##5##6##7##8##9 }
2247     \else: { \c_false_bool }
2248     \fi:
2249   }
2250   {#1}
2251 }
2252 \cs_set_protected:Npn \_cs_parm_from_arg_count_test:nnF #1#2
```

```

2253 {
2254   \if_meaning:w \c_false_bool #1
2255   \exp_after:wN \use_ii:nn
2256   \else:
2257     \exp_after:wN \use_i:nn
2258   \fi:
2259   { #2 {#1} }
2260 }

```

(End definition for `__kernel_cs_parm_from_arg_count:nnF` and `__cs_parm_from_arg_count_test:nnF`.)

4.15 Defining functions from a given number of arguments

```

2261 <@@=cs>

```

`__cs_count_signature:N` Counting the number of tokens in the signature, *i.e.*, the number of arguments the function should take. Since this is not used in any time-critical function, we simply use `\tl_count:n` if there is a signature, otherwise `-1` arguments to signal an error. We need a variant form right away.

```

2262 \cs_new:Npn \__cs_count_signature:N #1
2263 { \exp_args:Nf \__cs_count_signature:n { \cs_split_function:N #1 } }
2264 \cs_new:Npn \__cs_count_signature:n #1
2265 { \int_eval:n { \__cs_count_signature:nnN #1 } }
2266 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
2267 {
2268   \if_meaning:w \c_true_bool #3
2269   \tl_count:n {#2}
2270   \else:
2271     -1
2272   \fi:
2273 }
2274 \cs_new:Npn \__cs_count_signature:c
2275 { \exp_args:Nc \__cs_count_signature:N }

```

(End definition for `__cs_count_signature:N`, `__cs_count_signature:n`, and `__cs_count_signature:nnN`.)

`\cs_generate_from_arg_count:NNnn` We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since \TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```

2276 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
2277 {
2278   \__kernel_cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
2279   {
2280     \__kernel_msg_error:nnxx { kernel } { bad-number-of-arguments }
2281     { \token_to_str:N #1 } { \int_eval:n {#3} }
2282     \use_none:n
2283   }
2284   {#4}
2285 }

```


A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```

2286 \cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
2287 { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
2288 \cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
2289 { \exp_args:NNc \cs_generate_from_arg_count:NNnn }

```

(End definition for `\cs_generate_from_arg_count:NNnn`. This function is documented on page 13.)

4.16 Using the signature to define functions

```

2290 <@@=cs>

```

We can now combine some of the tools we have to provide a simple interface for defining functions, where the number of arguments is read from the signature. For instance, `\cs_set:Nn \foo_bar:nn {#1,#2}`.

We want to define `\cs_set:Nn` as

```

\cs_set:Nn
\cs_set:Nx
\cs_set_nopar:Nn
\cs_set_nopar:Nx
\cs_set_protected:Nn
\cs_set_protected:Nx
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Nx
\cs_gset:Nn
\cs_gset:Nx
\cs_gset_nopar:Nn
\cs_gset_nopar:Nx
\cs_gset_protected:Nn
\cs_gset_protected:Nx
\cs_new:Nn
\cs_new:Nx
\cs_new_nopar:Nn
\cs_new_nopar:Nx
\cs_new_protected:Nn
\cs_new_protected:Nx
\cs_new_protected_nopar:Nn
\cs_new_protected_nopar:Nx

```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```

2291 \cs_set:Npn \__cs_tmp:w #1#2#3
2292 {
2293   \cs_new_protected:cpx { cs_ #1 : #2 }
2294   {
2295     \exp_not:N \__cs_generate_from_signature:NNn
2296     \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
2297   }
2298 }
2299 \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
2300 {
2301   \use:x
2302   {
2303     \__cs_generate_from_signature:nnNNn
2304     \cs_split_function:N #2
2305   }
2306   #1 #2
2307 }
2308 \cs_new_protected:Npn \__cs_generate_from_signature:nnNNn #1#2#3#4#5#6
2309 {
2310   \bool_if:NTF #3
2311   {
2312     \str_if_eq_x:nnF { }
2313     { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
2314     {
2315       \__kernel_msg_error:nnx { kernel } { non-base-function }
2316       { \token_to_str:N #5 }
2317     }
2318   }

```

```

2318         \cs_generate_from_arg_count:NNnn
2319         #5 #4 { \tl_count:n {#2} } {#6}
2320     }
2321     {
2322         \__kernel_msg_error:nnx { kernel } { missing-colon }
2323         { \token_to_str:N #5 }
2324     }
2325 }
2326 \cs_new:Npn \__cs_generate_from_signature:n #1
2327 {
2328     \if:w n #1 \else: \if:w N #1 \else:
2329     \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
2330 }

```

Then we define the 24 variants beginning with N.

```

2331 \__cs_tmp:w { set } { Nn } { Npn }
2332 \__cs_tmp:w { set } { Nx } { Npx }
2333 \__cs_tmp:w { set_nopar } { Nn } { Npn }
2334 \__cs_tmp:w { set_nopar } { Nx } { Npx }
2335 \__cs_tmp:w { set_protected } { Nn } { Npn }
2336 \__cs_tmp:w { set_protected } { Nx } { Npx }
2337 \__cs_tmp:w { set_protected_nopar } { Nn } { Npn }
2338 \__cs_tmp:w { set_protected_nopar } { Nx } { Npx }
2339 \__cs_tmp:w { gset } { Nn } { Npn }
2340 \__cs_tmp:w { gset } { Nx } { Npx }
2341 \__cs_tmp:w { gset_nopar } { Nn } { Npn }
2342 \__cs_tmp:w { gset_nopar } { Nx } { Npx }
2343 \__cs_tmp:w { gset_protected } { Nn } { Npn }
2344 \__cs_tmp:w { gset_protected } { Nx } { Npx }
2345 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
2346 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
2347 \__cs_tmp:w { new } { Nn } { Npn }
2348 \__cs_tmp:w { new } { Nx } { Npx }
2349 \__cs_tmp:w { new_nopar } { Nn } { Npn }
2350 \__cs_tmp:w { new_nopar } { Nx } { Npx }
2351 \__cs_tmp:w { new_protected } { Nn } { Npn }
2352 \__cs_tmp:w { new_protected } { Nx } { Npx }
2353 \__cs_tmp:w { new_protected_nopar } { Nn } { Npn }
2354 \__cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End definition for \cs_set:Nn and others. These functions are documented on page 12.)

\cs_set:cn The 24 c variants simply use \exp_args:Nc.

\cs_set:cx

\cs_set_nopar:cn

\cs_set_nopar:cx

\cs_set_protected:cn

\cs_set_protected:cx

\cs_set_protected_nopar:cn

\cs_set_protected_nopar:cx

\cs_gset:cn

\cs_gset:cx

\cs_gset_nopar:cn

\cs_gset_nopar:cx

\cs_gset_protected:cn

\cs_gset_protected:cx

\cs_gset_protected_nopar:cn

\cs_gset_protected_nopar:cx

\cs_new:cn

\cs_new:cx

\cs_new_nopar:cn

\cs_new_nopar:cx

\cs_new_protected:cn

\cs_new_protected:cx

```

2355 \cs_set:Npn \__cs_tmp:w #1#2
2356 {
2357     \cs_new_protected:cpx { cs_ #1 : c #2 }
2358     {
2359         \exp_not:N \exp_args:Nc
2360         \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
2361     }
2362 }
2363 \__cs_tmp:w { set } { n }
2364 \__cs_tmp:w { set } { x }
2365 \__cs_tmp:w { set_nopar } { n }
2366 \__cs_tmp:w { set_nopar } { x }

```

```

2367 \__cs_tmp:w { set_protected } { n }
2368 \__cs_tmp:w { set_protected } { x }
2369 \__cs_tmp:w { set_protected_nopar } { n }
2370 \__cs_tmp:w { set_protected_nopar } { x }
2371 \__cs_tmp:w { gset } { n }
2372 \__cs_tmp:w { gset } { x }
2373 \__cs_tmp:w { gset_nopar } { n }
2374 \__cs_tmp:w { gset_nopar } { x }
2375 \__cs_tmp:w { gset_protected } { n }
2376 \__cs_tmp:w { gset_protected } { x }
2377 \__cs_tmp:w { gset_protected_nopar } { n }
2378 \__cs_tmp:w { gset_protected_nopar } { x }
2379 \__cs_tmp:w { new } { n }
2380 \__cs_tmp:w { new } { x }
2381 \__cs_tmp:w { new_nopar } { n }
2382 \__cs_tmp:w { new_nopar } { x }
2383 \__cs_tmp:w { new_protected } { n }
2384 \__cs_tmp:w { new_protected } { x }
2385 \__cs_tmp:w { new_protected_nopar } { n }
2386 \__cs_tmp:w { new_protected_nopar } { x }

```

(End definition for `\cs_set:Nn`. This function is documented on page 12.)

4.17 Checking control sequence equality

`\cs_if_eq_p:NN` Check if two control sequences are identical.

```

\cs_if_eq_p:cN 2387 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
\cs_if_eq_p:Nc 2388 {
\cs_if_eq_p:cc 2389   \if_meaning:w #1#2
\cs_if_eq:NNTF 2390   \prg_return_true: \else: \prg_return_false: \fi:
\cs_if_eq:cNTF 2391 }
\cs_if_eq:NcTF 2392 \cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
\cs_if_eq:NcTF 2393 \cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
\cs_if_eq:ccTF 2394 \cs_new:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNTF }
\cs_if_eq:ccTF 2395 \cs_new:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
\cs_if_eq:pNc 2396 \cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
\cs_if_eq:NcTF 2397 \cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
\cs_if_eq:NcT 2398 \cs_new:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNTF }
\cs_if_eq:NcF 2399 \cs_new:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
\cs_if_eq:pcc 2400 \cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
\cs_if_eq:ccTF 2401 \cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
\cs_if_eq:ccT 2402 \cs_new:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNTF }
\cs_if_eq:ccF 2403 \cs_new:Npn \cs_if_eq:ccF { \exp_args:Ncc \cs_if_eq:NNF }

```

(End definition for `\cs_if_eq:NNTF`. This function is documented on page 20.)

4.18 Diagnostic functions

```

2404 <@@=kernel>
\__kernel_chk_defined:NT Error if the variable #1 is not defined.
2405 \cs_new_protected:Npn \__kernel_chk_defined:NT #1#2
2406 {
2407   \cs_if_exist:NTF #1

```

```

2408     {#2}
2409     {
2410         \__kernel_msg_error:nxx { kernel } { variable-not-defined }
2411         { \token_to_str:N #1 }
2412     }
2413 }

```

(End definition for __kernel_chk_defined:NT.)

__kernel_register_show:N Simply using the \showthe primitive does not allow for line-wrapping, so instead use \tl_show:n and \tl_log:n (defined in l3tl and that performs line-wrapping). This displays $\sim\langle variable \rangle = \langle value \rangle$. We expand the value before-hand as otherwise some integers (such as \currentgrouplevel or \currentgrouptype) altered by the line-wrapping code would show wrong values.

```

\__kernel_register_show:c
\__kernel_register_log:N
\__kernel_register_log:c
\__kernel_register_show_aux:NN
\__kernel_register_show_aux:nnN
2414 \cs_new_protected:Npn \__kernel_register_show:N
2415 { \__kernel_register_show_aux:NN \tl_show:n }
2416 \cs_new_protected:Npn \__kernel_register_show:c
2417 { \exp_args:Nc \__kernel_register_show:N }
2418 \cs_new_protected:Npn \__kernel_register_log:N
2419 { \__kernel_register_show_aux:NN \tl_log:n }
2420 \cs_new_protected:Npn \__kernel_register_log:c
2421 { \exp_args:Nc \__kernel_register_log:N }
2422 \cs_new_protected:Npn \__kernel_register_show_aux:NN #1#2
2423 {
2424     \__kernel_chk_defined:NT #2
2425     {
2426         \exp_args:No \__kernel_register_show_aux:nnN
2427         { \tex_the:D #2 } #2 #1
2428     }
2429 }
2430 \cs_new_protected:Npn \__kernel_register_show_aux:nnN #1#2#3
2431 { \exp_args:No #3 { \token_to_str:N #2 = #1 } }

```

(End definition for __kernel_register_show:N and others.)

\cs_show:N Some control sequences have a very long name or meaning. Thus, simply using TeX's primitive \show could lead to overlong lines. The output of this primitive is mimicked to some extent, then the re-built string is given to \tl_show:n or \tl_log:n for line-wrapping. We must expand the meaning before passing it to the wrapping code as otherwise we would wrongly see the definitions that are in place there. To get correct escape characters, set the \escapechar in a group; this also localizes the assignment performed by x-expansion. The \cs_show:c and \cs_log:c commands convert their argument to a control sequence within a group to avoid showing \relax for undefined control sequences.

```

2432 \cs_new_protected:Npn \cs_show:N { \__kernel_show:NN \tl_show:n }
2433 \cs_new_protected:Npn \cs_show:c
2434 { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
2435 \cs_new_protected:Npn \cs_log:N { \__kernel_show:NN \tl_log:n }
2436 \cs_new_protected:Npn \cs_log:c
2437 { \group_begin: \exp_args:NNc \group_end: \cs_log:N }
2438 \cs_new_protected:Npn \__kernel_show:NN #1#2
2439 {
2440     \group_begin:

```

```

2441     \int_set:Nn \tex_escapechar:D { ‘\ }
2442     \exp_args:NNx
2443     \group_end:
2444     #1 { \token_to_str:N #2 = \cs_meaning:N #2 }
2445 }

```

(End definition for `\cs_show:N`, `\cs_log:N`, and `__kernel_show:NN`. These functions are documented on page 15.)

4.19 Doing nothing functions

`\prg_do_nothing:` This does not fit anywhere else!

```

2446 \cs_new:Npn \prg_do_nothing: { }

```

(End definition for `\prg_do_nothing:`. This function is documented on page 8.)

4.20 Breaking out of mapping functions

```

2447 <@@=prg>

```

`\prg_break_point:Nn` In inline mappings, the nesting level must be reset at the end of the mapping, even when the user decides to break out. This is done by putting the code that must be performed as an argument of `__prg_break_point:Nn`. The breaking functions are then defined to jump to that point and perform the argument of `__prg_break_point:Nn`, before the user’s code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

```

2448 \cs_new_eq:NN \prg_break_point:Nn \use_ii:nn
2449 \cs_new:Npn \prg_map_break:Nn #1#2#3 \prg_break_point:Nn #4#5
2450 {
2451     #5
2452     \if_meaning:w #1 #4
2453     \exp_after:wN \use_iii:nnn
2454     \fi:
2455     \prg_map_break:Nn #1 {#2}
2456 }

```

(End definition for `\prg_break_point:Nn` and `\prg_map_break:Nn`. These functions are documented on page 100.)

`\prg_break_point:` Very simple analogues of `\prg_break_point:Nn` and `\prg_map_break:Nn`, for use in fast short-term recursions which are not mappings, do not need to support nesting, and in which nothing has to be done at the end of the loop.

```

2457 \cs_new_eq:NN \prg_break_point: \prg_do_nothing:
2458 \cs_new:Npn \prg_break: #1 \prg_break_point: { }
2459 \cs_new:Npn \prg_break:n #1#2 \prg_break_point: {#1}

```

(End definition for `\prg_break_point:`, `\prg_break:`, and `\prg_break:n`. These functions are documented on page 100.)

```

2460 </initex | package>

```

5 l3expan implementation

2461 `*initex | package)`

2462 `\@@=exp)`

`\l__exp_internal_tl` The `\exp_` module has its private variable to temporarily store the result of `x`-type argument expansion. This is done to avoid interference with other functions using temporary variables.

(End definition for \l__exp_internal_tl.)

`\exp_after:wN` These are defined in `l3basics`, as they are needed “early”. This is just a reminder of that fact!

`\exp_not:N`

`\exp_not:n`

(End definition for \exp_after:wN, \exp_not:N, and \exp_not:n. These functions are documented on page 30.)

5.1 General expansion

In this section a general mechanism for defining functions that handle arguments is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the L^AT_EX3 names for `\cs_set:Npx` at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 5.3. In section 5.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l__exp_internal_tl` This scratch token list variable is defined in `l3basics`.

(End definition for \l__exp_internal_tl.)

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are `long` since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\::⟨Z⟩` always has signature `#1\:::#2#3` where `#1` holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, `#2` is the carried over result of the previous expansion steps and `#3` is the argument about to be processed. One exception to this rule is `\::p`, which has to grab an argument delimited by a left brace.

`__exp_arg_next:nnn` `#1` is the result of an expansion step, `#2` is the remaining argument manipulations and `#3` is the current result of the expansion chain. This auxiliary function moves `#1` back after `#3` in the input stream and checks if any expansion is left to be done by calling `#2`. In by far the most cases we need to add a set of braces to the result of an argument manipulation so it is more effective to do it directly here. Actually, so far only the `c` of the final argument manipulation variants does not require a set of braces.

2463 `\cs_new:Npn __exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }`

2464 `\cs_new:Npn __exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }`

(End definition for __exp_arg_next:nnn and __exp_arg_next:Nnn.)

`\:::` The end marker is just another name for the identity function.

2465 `\cs_new:Npn \::: #1 {#1}`

(End definition for \:::. This function is documented on page 33.)

\::n This function is used to skip an argument that doesn't need to be expanded.

```
2466 \cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(End definition for \::n. This function is documented on page 33.)

\::N This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
2467 \cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(End definition for \::N. This function is documented on page 33.)

\::p This function is used to skip an argument that is delimited by a left brace and doesn't need to be expanded. It is not wrapped in braces in the result.

```
2468 \cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(End definition for \::p. This function is documented on page 33.)

\::c This function is used to skip an argument that is turned into a control sequence without expansion.

```
2469 \cs_new:Npn \::c #1 \::: #2#3
2470 { \exp_after:wN \_exp_arg_next:nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(End definition for \::c. This function is documented on page 33.)

\::o This function is used to expand an argument once.

```
2471 \cs_new:Npn \::o #1 \::: #2#3
2472 { \exp_after:wN \_exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(End definition for \::o. This function is documented on page 33.)

\::f This function is used to expand a token list until the first unexpandable token is found. This is achieved through `\exp:w \exp_end_continue_f:w` that expands everything in its way following it. This scanning procedure is terminated once the expansion hits something non-expandable (if that is a space it is removed). We introduce `\exp_stop_f:` to mark such an end-of-expansion marker. For example, `f`-expanding `\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }` where `\l_tmpa_tl` contains the characters `lur` gives `\tex_let:D \aaa = \blurb` which then turns out to start with the non-expandable token `\tex_let:D`. Since the expansion of `\exp:w \exp_end_continue_f:w` is empty, we wind up with a fully expanded list, only TeX has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```
2473 \cs_new:Npn \::f #1 \::: #2#3
2474 {
2475   \exp_after:wN \_exp_arg_next:nnn
2476   \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2477   {#1} {#2}
2478 }
2479 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(End definition for \::f and \exp_stop_f:. These functions are documented on page 33.)

\::x This function is used to expand an argument fully. We build in the expansion of `__exp_arg_next:nnn`.

```

2480 \cs_new_protected:Npn \::x #1 \::: #2#3
2481 {
2482   \cs_set_nopar:Npx \l__exp_internal_tl
2483   { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } }
2484   \l__exp_internal_tl
2485 }

```

(End definition for `\::x`. This function is documented on page 33.)

\::v These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim`, **`\::V`** `muskip`, or built-in TeX register. The `V` version expects a single token whereas `v` like `c` creates a cname from its argument given in braces and then evaluates it as if it was a `V`. The `\exp:w` sets off an expansion similar to an `f`-type expansion, which we terminate using `\exp_end:.` The argument is returned in braces.

```

2486 \cs_new:Npn \::V #1 \::: #2#3
2487 {
2488   \exp_after:wN \__exp_arg_next:nnn
2489   \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2490   {#1} {#2}
2491 }
2492 \cs_new:Npn \::v # 1\::: #2#3
2493 {
2494   \exp_after:wN \__exp_arg_next:nnn
2495   \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2496   {#1} {#2}
2497 }

```

(End definition for `\::v` and `\::V`. These functions are documented on page 33.)

`__exp_eval_register:N`
`__exp_eval_register:c`
`__exp_eval_error_msg:w`

This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in TeX register such as `\count`. For the TeX registers we have to utilize a `\the` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\the` and when not to. What we want here is to find out whether the token expands to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the token in question when it has been prefixed with `\exp_not:N` and the token itself. If it is a macro, the prefixed `\exp_not:N` temporarily turns it into the primitive `\scan_stop:.`

```

2498 \cs_new:Npn \__exp_eval_register:N #1
2499 {
2500   \exp_after:wN \if_meaning:w \exp_not:N #1 #1

```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:.` In that case we throw an error. We could let TeX do it for us but that would result in the rather obscure

! You can't use '`\relax`' after `\the`.

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```

2501   \if_meaning:w \scan_stop: #1
2502   \__exp_eval_error_msg:w
2503   \fi:

```


The next bit requires some explanation. The function must be initiated by `\exp:w` and we want to terminate this expansion chain by inserting the `\exp_end:` token. However, we have to expand the register `#1` before we do that. If it is a `TeX` register, we need to execute the sequence `\exp_after:wN \exp_end: \tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN \exp_end: #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```

2504     \else:
2505         \exp_after:wN \use_i_ii:nnn
2506     \fi:
2507     \exp_after:wN \exp_end: \tex_the:D #1
2508 }
2509 \cs_new:Npn \__exp_eval_register:c #1
2510 { \exp_after:wN \__exp_eval_register:N \cs:w #1 \cs_end: }

```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```

! Undefined control sequence.
<argument> \LaTeX3 error:
                               Erroneous variable used!
1.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}

2511 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
2512 {
2513     \fi:
2514     \fi:
2515     \__kernel_msg_expandable_error:nnn { kernel } { bad-variable } {#2}
2516     \exp_end:
2517 }

```

(End definition for `__exp_eval_register:N` and `__exp_eval_error_msg:w`.)

5.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable.

`\exp_args:Nc` In `l3basics`.

`\exp_args:cc`

(End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 26.)

`\exp_args:NNc`

`\exp_args:Ncc`

`\exp_args:Nccc`

Here are the functions that turn their argument into csnames but are expandable.

```

2518 \cs_new:Npn \exp_args:NNc #1#2#3
2519 { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3 \cs_end: }
2520 \cs_new:Npn \exp_args:Ncc #1#2#3
2521 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
2522 \cs_new:Npn \exp_args:Nccc #1#2#3#4
2523 {
2524     \exp_after:wN #1
2525     \cs:w #2 \exp_after:wN \cs_end:
2526     \cs:w #3 \exp_after:wN \cs_end:
2527     \cs:w #4 \cs_end:
2528 }

```

(End definition for `\exp_args:NNc`, `\exp_args:Ncc`, and `\exp_args:Nccc`. These functions are documented on page 27.)

`\exp_args:No` Those lovely runs of expansion!
`\exp_args:NNo` 2529 `\cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }`
`\exp_args:NNNo` 2530 `\cs_new:Npn \exp_args:NNo #1#2#3`
2531 `{ \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }`
2532 `\cs_new:Npn \exp_args:NNNo #1#2#3#4`
2533 `{ \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }`

(End definition for `\exp_args:No`, `\exp_args:NNo`, and `\exp_args:NNNo`. These functions are documented on page 26.)

`\exp_args:Nf`
`\exp_args:NV` 2534 `\cs_new:Npn \exp_args:Nf #1#2`
`\exp_args:Nv` 2535 `{ \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }`
2536 `\cs_new:Npn \exp_args:Nv #1#2`
2537 `{`
2538 `\exp_after:wN #1 \exp_after:wN`
2539 `{ \exp:w __exp_eval_register:c {#2} }`
2540 `}`
2541 `\cs_new:Npn \exp_args:NV #1#2`
2542 `{`
2543 `\exp_after:wN #1 \exp_after:wN`
2544 `{ \exp:w __exp_eval_register:N #2 }`
2545 `}`

(End definition for `\exp_args:Nf`, `\exp_args:NV`, and `\exp_args:Nv`. These functions are documented on page 27.)

`\exp_args:NNV` Some more hand-tuned function with three arguments. If we forced that an `o` argument
`\exp_args:NNv` always has braces, we could implement `\exp_args:Nco` with less tokens and only two
`\exp_args:NNf` arguments.

`\exp_args:Nco` 2546 `\cs_new:Npn \exp_args:NNV #1#2#3`
`\exp_args:NcV` 2547 `{`
`\exp_args:Ncv` 2548 `\exp_after:wN #1`
`\exp_args:Ncf` 2549 `\exp_after:wN #2`
`\exp_args:NVV` 2550 `\exp_after:wN { \exp:w __exp_eval_register:N #3 }`
2551 `}`
2552 `\cs_new:Npn \exp_args:NNv #1#2#3`
2553 `{`
2554 `\exp_after:wN #1`
2555 `\exp_after:wN #2`
2556 `\exp_after:wN { \exp:w __exp_eval_register:c {#3} }`
2557 `}`
2558 `\cs_new:Npn \exp_args:NNf #1#2#3`
2559 `{`
2560 `\exp_after:wN #1`
2561 `\exp_after:wN #2`
2562 `\exp_after:wN { \exp:w \exp_end_continue_f:w #3 }`
2563 `}`
2564 `\cs_new:Npn \exp_args:Nco #1#2#3`
2565 `{`
2566 `\exp_after:wN #1`
2567 `\cs:w #2 \exp_after:wN \cs_end:`
2568 `\exp_after:wN {#3}`
2569 `}`

```

2570 \cs_new:Npn \exp_args:NcV #1#2#3
2571 {
2572     \exp_after:wN #1
2573     \cs:w #2 \exp_after:wN \cs_end:
2574     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2575 }
2576 \cs_new:Npn \exp_args:Ncv #1#2#3
2577 {
2578     \exp_after:wN #1
2579     \cs:w #2 \exp_after:wN \cs_end:
2580     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2581 }
2582 \cs_new:Npn \exp_args:Ncf #1#2#3
2583 {
2584     \exp_after:wN #1
2585     \cs:w #2 \exp_after:wN \cs_end:
2586     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2587 }
2588 \cs_new:Npn \exp_args:NVV #1#2#3
2589 {
2590     \exp_after:wN #1
2591     \exp_after:wN { \exp:w \exp_after:wN
2592         \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
2593     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2594 }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page 27.)

`\exp_args:NNNV` A few more that we can hand-tune.

```

\exp_args:NcNc 2595 \cs_new:Npn \exp_args:NNNV #1#2#3#4
\exp_args:NcNo 2596 {
\exp_args:Ncco 2597     \exp_after:wN #1
2598     \exp_after:wN #2
2599     \exp_after:wN #3
2600     \exp_after:wN { \exp:w \__exp_eval_register:N #4 }
2601 }
2602 \cs_new:Npn \exp_args:NcNc #1#2#3#4
2603 {
2604     \exp_after:wN #1
2605     \cs:w #2 \exp_after:wN \cs_end:
2606     \exp_after:wN #3
2607     \cs:w #4 \cs_end:
2608 }
2609 \cs_new:Npn \exp_args:NcNo #1#2#3#4
2610 {
2611     \exp_after:wN #1
2612     \cs:w #2 \exp_after:wN \cs_end:
2613     \exp_after:wN #3
2614     \exp_after:wN {#4}
2615 }
2616 \cs_new:Npn \exp_args:Ncco #1#2#3#4
2617 {
2618     \exp_after:wN #1
2619     \cs:w #2 \exp_after:wN \cs_end:

```

```

2620     \cs:w #3 \exp_after:wN \cs_end:
2621     \exp_after:wN {#4}
2622 }

```

(End definition for `\exp_args:NNNV` and others. These functions are documented on page 28.)

5.3 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions actually take no arguments themselves.

`\exp_args:Nx`

```

2623 \cs_new_protected:Npn \exp_args:Nx { \::x \:: }

```

(End definition for `\exp_args:Nx`. This function is documented on page 27.)

`\exp_args:Nnc` Here are the actual function definitions, using the helper functions above.

```

\exp_args:Nno 2624 \cs_new:Npn \exp_args:Nnc { \::n \::c \:: }
\exp_args:NnV 2625 \cs_new:Npn \exp_args:Nno { \::n \::o \:: }
\exp_args:NnV 2626 \cs_new:Npn \exp_args:NnV { \::n \::V \:: }
\exp_args:Nnf 2627 \cs_new:Npn \exp_args:NnV { \::n \::v \:: }
\exp_args:Noc 2628 \cs_new:Npn \exp_args:Nnf { \::n \::f \:: }
\exp_args:Noo 2629 \cs_new:Npn \exp_args:Noc { \::o \::c \:: }
\exp_args:Nof 2630 \cs_new:Npn \exp_args:Noo { \::o \::o \:: }
\exp_args:NVo 2631 \cs_new:Npn \exp_args:Nof { \::o \::f \:: }
\exp_args:Nfo 2632 \cs_new:Npn \exp_args:NVo { \::V \::o \:: }
\exp_args:Nff 2633 \cs_new:Npn \exp_args:Nfo { \::f \::o \:: }
\exp_args:Nff 2634 \cs_new:Npn \exp_args:Nff { \::f \::f \:: }
\exp_args:NNx 2635 \cs_new_protected:Npn \exp_args:NNx { \::N \::x \:: }
\exp_args:Ncx 2636 \cs_new_protected:Npn \exp_args:Ncx { \::c \::x \:: }
\exp_args:Nnx 2637 \cs_new_protected:Npn \exp_args:Nnx { \::n \::x \:: }
\exp_args:Nox 2638 \cs_new_protected:Npn \exp_args:Nox { \::o \::x \:: }
\exp_args:Nxo 2639 \cs_new_protected:Npn \exp_args:Nxo { \::x \::o \:: }
\exp_args:Nxx 2640 \cs_new_protected:Npn \exp_args:Nxx { \::x \::x \:: }

```

(End definition for `\exp_args:Nnc` and others. These functions are documented on page 28.)

`\exp_args:NNcf`

```

\exp_args:NNno 2641 \cs_new:Npn \exp_args:NNcf { \::N \::c \::f \:: }
\exp_args:NNnV 2642 \cs_new:Npn \exp_args:NNno { \::N \::n \::o \:: }
\exp_args:NNoo 2643 \cs_new:Npn \exp_args:NNnV { \::N \::n \::V \:: }
\exp_args:NNVV 2644 \cs_new:Npn \exp_args:NNoo { \::N \::o \::o \:: }
\exp_args:Ncno 2645 \cs_new:Npn \exp_args:NNVV { \::N \::V \::V \:: }
\exp_args:NcnV 2646 \cs_new:Npn \exp_args:Ncno { \::c \::n \::o \:: }
\exp_args:Ncoo 2647 \cs_new:Npn \exp_args:NcnV { \::c \::n \::V \:: }
\exp_args:NcVV 2648 \cs_new:Npn \exp_args:Ncoo { \::c \::o \::o \:: }
\exp_args:Nnnc 2649 \cs_new:Npn \exp_args:NcVV { \::c \::V \::V \:: }
\exp_args:Nnno 2650 \cs_new:Npn \exp_args:Nnnc { \::n \::n \::c \:: }
\exp_args:Nnnf 2651 \cs_new:Npn \exp_args:Nnno { \::n \::n \::o \:: }
\exp_args:Nnff 2652 \cs_new:Npn \exp_args:Nnnf { \::n \::n \::f \:: }
\exp_args:Nooo 2653 \cs_new:Npn \exp_args:Nnff { \::n \::f \::f \:: }
\exp_args:Noof 2654 \cs_new:Npn \exp_args:Nooo { \::o \::o \::o \:: }
\exp_args:Nffo 2655 \cs_new:Npn \exp_args:Noof { \::o \::o \::f \:: }
\exp_args:NNNx 2656 \cs_new:Npn \exp_args:Nffo { \::f \::f \::o \:: }
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox

```

```

2657 \cs_new_protected:Npn \exp_args:NNNx { \::N \::N \::x \:: }
2658 \cs_new_protected:Npn \exp_args:NNnx { \::N \::n \::x \:: }
2659 \cs_new_protected:Npn \exp_args:NNox { \::N \::o \::x \:: }
2660 \cs_new_protected:Npn \exp_args:Nnnx { \::n \::n \::x \:: }
2661 \cs_new_protected:Npn \exp_args:Nnox { \::n \::o \::x \:: }
2662 \cs_new_protected:Npn \exp_args:Nccx { \::c \::c \::x \:: }
2663 \cs_new_protected:Npn \exp_args:Ncnx { \::c \::n \::x \:: }
2664 \cs_new_protected:Npn \exp_args:Noox { \::o \::o \::x \:: }

```

(End definition for `\exp_args:NNcf` and others. These functions are documented on page 28.)

5.4 Last-unbraced versions

`__exp_arg_last_unbraced:nn` There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

\::o_unbraced
\::V_unbraced
\::v_unbraced
\::f_unbraced
\::x_unbraced
2665 \cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
2666 \cs_new:Npn \::o_unbraced \:: #1#2
2667 { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
2668 \cs_new:Npn \::V_unbraced \:: #1#2
2669 {
2670   \exp_after:wN \__exp_arg_last_unbraced:nn
2671   \exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1}
2672 }
2673 \cs_new:Npn \::v_unbraced \:: #1#2
2674 {
2675   \exp_after:wN \__exp_arg_last_unbraced:nn
2676   \exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1}
2677 }
2678 \cs_new:Npn \::f_unbraced \:: #1#2
2679 {
2680   \exp_after:wN \__exp_arg_last_unbraced:nn
2681   \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
2682 }
2683 \cs_new_protected:Npn \::x_unbraced \:: #1#2
2684 {
2685   \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
2686   \l__exp_internal_tl
2687 }

```

(End definition for `__exp_arg_last_unbraced:nn` and others. These functions are documented on page 33.)

`\exp_last_unbraced:No` Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

\exp_last_unbraced:Nv
\exp_last_unbraced:Nf
\exp_last_unbraced:NNo
\exp_last_unbraced:NNv
\exp_last_unbraced:Nco
\exp_last_unbraced:NcV
\exp_last_unbraced:NNNo
\exp_last_unbraced:NNNV
\exp_last_unbraced:Nno
\exp_last_unbraced:Noo
\exp_last_unbraced:Nfo
\exp_last_unbraced:NnNo
\exp_last_unbraced:NNNNo
\exp_last_unbraced:Nx
2688 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
2689 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2690 { \exp_after:wN #1 \exp:w \__exp_eval_register:N #2 }
2691 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2692 { \exp_after:wN #1 \exp:w \__exp_eval_register:c {#2} }
2693 \cs_new:Npn \exp_last_unbraced:Nf #1#2
2694 { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
2695 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
2696 { \exp_after:wN #1 \exp_after:wN #2 #3 }
2697 \cs_new:Npn \exp_last_unbraced:NNv #1#2#3

```

```

2698 {
2699   \exp_after:wN #1
2700   \exp_after:wN #2
2701   \exp:w \_\_exp_eval_register:N #3
2702 }
2703 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
2704 { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
2705 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
2706 {
2707   \exp_after:wN #1
2708   \cs:w #2 \exp_after:wN \cs_end:
2709   \exp:w \_\_exp_eval_register:N #3
2710 }
2711 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
2712 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
2713 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
2714 {
2715   \exp_after:wN #1
2716   \exp_after:wN #2
2717   \exp_after:wN #3
2718   \exp:w \_\_exp_eval_register:N #4
2719 }
2720 \cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \:: }
2721 \cs_new:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \:: }
2722 \cs_new:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \:: }
2723 \cs_new:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \:: }
2724 \cs_new:Npn \exp_last_unbraced:NNNNo #1#2#3#4#5
2725 { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 \exp_after:wN #4 #5 }
2726 \cs_new_protected:Npn \exp_last_unbraced:Nx { \::x_unbraced \:: }

```

(End definition for `\exp_last_unbraced:No` and others. These functions are documented on page 29.)

If #2 is a single token then this can be implemented as

```

\exp_last_two_unbraced:Noo If #2 is a single token then this can be implemented as
\_\_exp_last_two_unbraced:noN
\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
{ \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }

```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```

2727 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
2728 { \exp_after:wN \_\_exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
2729 \cs_new:Npn \_\_exp_last_two_unbraced:noN #1#2#3
2730 { \exp_after:wN #3 #2 #1 }

```

(End definition for `\exp_last_two_unbraced:Noo` and `__exp_last_two_unbraced:noN`. This function is documented on page 29.)

5.5 Preventing expansion

`__kernel_exp_not:w` At the kernel level, we need the primitive behaviour to allow expansion *before* the brace group.

```

2731 \cs_new_eq:NN \_\_kernel_exp_not:w \etex_unexpanded:D

```

(End definition for `__kernel_exp_not:w`.)

```

\exp_not:c All these except \exp_not:c call the kernel-internal \__kernel_exp_not:w namely
\exp_not:o \etex_unexpanded:D.
\exp_not:f 2732 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
\exp_not:V 2733 \cs_new:Npn \exp_not:o #1 { \__kernel_exp_not:w \exp_after:wN {#1} }
2734 \cs_new:Npn \exp_not:f #1
2735 { \etex_unexpanded:D \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
2736 \cs_new:Npn \exp_not:V #1
2737 {
2738   \__kernel_exp_not:w \exp_after:wN
2739   { \exp:w \__exp_eval_register:N #1 }
2740 }
2741 \cs_new:Npn \exp_not:v #1
2742 {
2743   \__kernel_exp_not:w \exp_after:wN
2744   { \exp:w \__exp_eval_register:c {#1} }
2745 }

```

(End definition for `\exp_not:c` and others. These functions are documented on page 30.)

5.6 Controlled expansion

`\exp:w` To trigger a sequence of “arbitrarily” many expansions we need a method to invoke TeX’s expansion mechanism in such a way that (a) we are able to stop it in a controlled manner and (b) the result of what triggered the expansion in the first place is null, i.e., that we do not get any unwanted side effects. There aren’t that many possibilities in TeX; in fact the one explained below might well be the only one (as normally the result of expansion is not null).

The trick here is to make use of the fact that `\tex_romannumeral:D` expands the tokens following it when looking for a number and that its expansion is null if that number turns out to be zero or negative. So we use that to start the expansion sequence: `\exp:w` is set equal to `\tex_romannumeral:D` in `l3basics`. To stop the expansion sequence in a controlled way all we need to provide is a constant integer zero as part of expanded tokens. As this is an integer constant it immediately stops `\tex_romannumeral:D`’s search for a number. Again, the definition of `\exp_end:` as the integer constant zero is in `l3basics`. (Note that according to our specification all tokens we expand initiated by `\exp:w` are supposed to be expandable (as well as their replacement text in the expansion) so we will not encounter a “number” that actually result in a roman numeral being generated. Or if we do then the programmer made a mistake.)

If on the other hand we want to stop the initial expansion sequence but continue with an f-type expansion we provide the alphabetic constant `’^^@` that also represents 0 but this time TeX’s syntax for a *<number>* continues searching for an optional space (and it continues expansion doing that) — see TeXbook page 269 for details.

```

2746 \group_begin:
2747   \tex_catcode:D ‘^^@ = 13
2748   \cs_new_protected:Npn \exp_end_continue_f:w { ‘^^@ }

```

If the above definition ever appears outside its proper context the active character `^^@` will be executed so we turn this into an error. The test for existence covers the (unlikely) case that some other code has already defined `^^@`: this is true for example for `xmltex.tex`.

```

2749   \if_cs_exist:N ^^@
2750   \else:
2751     \cs_new:Npn ^^@

```

```

2752         { \_kernel_msg_expandable_error:nn { kernel } { bad-exp-end-f } }
2753     \fi:

```

The same but grabbing an argument to remove spaces and braces.

```

2754     \cs_new:Npn \exp_end_continue_f:nw #1 { '^~@ #1 }
2755 \group_end:

```

(End definition for `\exp:w` and others. These functions are documented on page 32.)

5.7 Defining function variants

```

2756 <@@=cs>

```

```

\cs_generate_variant:Nn #1 : Base form of a function; e.g., \tl_set:Nn
\cs_generate_variant:cn #2 : One or more variant argument specifiers; e.g., {Nx,c,cx}

```

After making sure that the base form exists, test whether it is protected or not and define `_cs_tmp:w` as either `\cs_new:Npx` or `\cs_new_protected:Npx`, which is then used to define all the variants (except those involving x-expansion, always protected). Split up the original base function only once, to grab its name and signature. Then we wish to iterate through the comma list of variant argument specifiers, which we first convert to a string: the reason is explained later.

```

2757 \_kernel_patch:nnNNpn { \_kernel_chk_cs_exist:N #1 } { }
2758 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
2759 {
2760     \_cs_generate_variant:N #1
2761     \use:x
2762     {
2763         \_cs_generate_variant:nnNN
2764         \cs_split_function:N #1
2765         \exp_not:N #1
2766         \tl_to_str:n {#2} ,
2767         \exp_not:N \scan_stop: ,
2768         \exp_not:N \q_recursion_stop
2769     }
2770 }
2771 \cs_new_protected:Npn \cs_generate_variant:cn
2772 { \exp_args:Nc \cs_generate_variant:Nn }

```

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 24.)

```

\_cs_generate_variant:N
\_cs_generate_variant:ww
\_cs_generate_variant:wwNw

```

The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the `\else:` branch is safe because non-expandable primitives cannot be `TeX` conditionals.

The other case where variants should be protected is when the parent function is a protected macro: then `protected` appears in the meaning before the first occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We use `ma` rather than the full `macro` because the meaning of the `\firstmark` primitive (and four others) can contain an arbitrary string after a leading `firstmark:`. Then, look for `pr` in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, `\long_`, `\protected_`, `\protected\long_`, `\first`, `\top`, `\bot`, `\splittop`, or `\splitbot`, with `\` replaced by the appropriate escape character. If

`pr` appears in the part before `ma`, the first `\q_mark` is taken as an argument of the `wwNw` auxiliary, and `#3` is `\cs_new_protected:Npx`, otherwise it is `\cs_new:Npx`.

```

2773 \cs_new_protected:Npx \__cs_generate_variant:N #1
2774 {
2775   \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
2776   \exp_not:N \exp_not:N #1 #1
2777   \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx
2778   \exp_not:N \else:
2779   \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
2780   \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
2781   \exp_not:N \q_mark
2782   \exp_not:N \q_mark \cs_new_protected:Npx
2783   \tl_to_str:n { pr }
2784   \exp_not:N \q_mark \cs_new:Npx
2785   \exp_not:N \q_stop
2786   \exp_not:N \fi:
2787 }
2788 \exp_last_unbraced:NNNNo
2789 \cs_new_protected:Npn \__cs_generate_variant:ww
2790 #1 { \tl_to_str:n { ma } } #2 \q_mark
2791 { \__cs_generate_variant:wwNw #1 }
2792 \exp_last_unbraced:NNNNo
2793 \cs_new_protected:Npn \__cs_generate_variant:wwNw
2794 #1 { \tl_to_str:n { pr } } #2 \q_mark #3 #4 \q_stop
2795 { \cs_set_eq:NN \__cs_tmp:w #3 }

```

(End definition for `__cs_generate_variant:N`, `__cs_generate_variant:ww`, and `__cs_generate_variant:wwNw`.)

`__cs_generate_variant:nnNN` #1 : Base name.
 #2 : Base signature.
 #3 : Boolean.
 #4 : Base function.

If the boolean is `\c_false_bool`, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as `#4` for efficiency.

```

2796 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
2797 {
2798   \if_meaning:w \c_false_bool #3
2799   \__kernel_msg_error:nnx { kernel } { missing-colon }
2800   { \token_to_str:c {#1} }
2801   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2802   \fi:
2803   \__cs_generate_variant:Nnnw #4 {#1}{#2}
2804 }

```

(End definition for `__cs_generate_variant:nnNN`.)

`__cs_generate_variant:Nnnw` #1 : Base function.
 #2 : Base name.
 #3 : Base signature.
 #4 : Beginning of variant signature.

First check whether to terminate the loop over variant forms. Then, for each variant form, construct a new function name using the original base name, the variant signature

consisting of l letters and the last $k - l$ letters of the base signature (of length k). For example, for a base function `\prop_put:Nnn` which needs a `cV` variant form, we want the new signature to be `cVn`.

There are further subtleties:

- In `\cs_generate_variant:Nn \foo:nnTF {xxTF}`, we must define `\foo:xxTF` using `\exp_args:Nxx`, rather than a hypothetical `\exp_args:NxxTF`. Thus, we wish to trim a common trailing part from the base signature and the variant signature.
- In `\cs_generate_variant:Nn \foo:on {ox}`, the function `\foo:ox` must be defined using `\exp_args:Nnx`, not `\exp_args:Nox`, to avoid double `o` expansion.
- Lastly, `\cs_generate_variant:Nn \foo:on {xn}` must trigger an error, because we do not have a means to replace `o`-expansion by `x`-expansion. More generally, we can only convert `N` to `c`, or convert `n` to `V`, `v`, `o`, `f`, `x`.

All this boils down to a few rules. Only `n` and `N`-type arguments can be replaced by `\cs_generate_variant:Nn`. Other argument types are allowed to be passed unchanged from the base form to the variant: in the process they are changed to `n` except for `N` and `p`-type arguments. A common trailing part is ignored.

We compare the base and variant signatures one character at a time within `x`-expansion. The result is given to `__cs_generate_variant:wwNN` (defined later) in the form `<processed variant signature> \q_mark <errors> \q_stop <base function> <new function>`. If all went well, `<errors>` is empty; otherwise, it is a kernel error message and some clean-up code.

Note the space after `#3` and after the following brace group. Those are ignored by \TeX when fetching the last argument for `__cs_generate_variant_loop:nNwN`, but can be used as a delimiter for `__cs_generate_variant_loop_end:nwwwNNnn`.

```

2805 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
2806 {
2807   \if_meaning:w \scan_stop: #4
2808   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2809   \fi:
2810   \use:x
2811   {
2812     \exp_not:N \__cs_generate_variant:wwNN
2813     \__cs_generate_variant_loop:nNwN { }
2814     #4
2815     \__cs_generate_variant_loop_end:nwwwNNnn
2816     \q_mark
2817     #3 ~
2818     { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
2819     { }
2820     \q_stop
2821     \exp_not:N #1 {#2} {#4}
2822   }
2823   \__cs_generate_variant:Nnnw #1 {#2} {#3}
2824 }
```

(End definition for `__cs_generate_variant:Nnnw`.)

<pre> __cs_generate_variant_loop:nNwN __cs_generate_variant_loop_base:N __cs_generate_variant_loop_same:w __cs_generate_variant_loop_end:nwwwNNnn __cs_generate_variant_loop_long:wNNnn __cs_generate_variant_loop_invalid:NNwNNnn __cs_generate_variant_loop_special:NNwNNnn __cs_generate_variant_loop_warning:nnxxx </pre>	<p>#1 : Last few consecutive letters common between the base and variant (more precisely, <code>__cs_generate_variant_same:N</code> <i><letter></i> for each letter).</p>
---	---

#2 : Next variant letter.
 #3 : Remainder of variant form.
 #4 : Next base letter.

The first argument is populated by `__cs_generate_variant_loop_same:w` when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed if the base is N and the variant is c, or when the base is n and the variant is o, V, v, f or x. Otherwise, call `__cs_generate_variant_loop_invalid:NNwNNnn` to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second argument of `__cs_generate_variant:wwNN`. If the letters are distinct and the base letter is indeed n or N, leave in the input stream whatever argument #1 was collected, and the next variant letter #2, then loop by calling `__cs_generate_variant_loop:nNwN`.

The loop can stop in three ways.

- If the end of the variant form is encountered first, #2 is `__cs_generate_variant_loop_end:nwwwNNnn` (expanded by the conditional `\if:w`), which inserts some tokens to end the conditional; grabs the *base name* as #7, the *variant signature* #8, the *next base letter* #1 and the part #3 of the base signature that wasn't read yet; and combines those into the *new function* to be defined.
- If the end of the base form is encountered first, #4 is `~{} \fi:` which ends the conditional (with an empty expansion), followed by `__cs_generate_variant_loop_long:wNNnn`, which places an error as the second argument of `__cs_generate_variant:wwNN`.
- The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is not the right one (n or N to support the variant). In that case too an error is placed as the second argument of `__cs_generate_variant:wwNN`.

Note that if the variant form has the same length as the base form, #2 is as described in the first point, and #4 as described in the second point above. The `__cs_generate_variant_loop_end:nwwwNNnn` breaking function takes the empty brace group in #4 as its first argument: this empty brace group produces the correct signature for the full variant.

Since people seem to have tried generating N or c-type variants of n-type arguments, and n, o, V, v, f, x variants of N-type arguments, in those cases we only produce a warning.

```

2825 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \q_mark #4
2826 {
2827   \if:w #2 #4
2828     \exp_after:wN \__cs_generate_variant_loop_same:w
2829   \else:
2830     \if:w #4 \__cs_generate_variant_loop_base:N #2 \else:
2831       \if:w 0
2832         \if:w N #4 \else: \if:w n #4 \else: 1 \fi: \fi:
2833         \if:w \scan_stop: \__cs_generate_variant_loop_base:N #2 1 \fi:
2834         0
2835         \__cs_generate_variant_loop_special:NNwNNnn #4#2
2836       \else:
2837         \__cs_generate_variant_loop_invalid:NNwNNnn #4#2

```

```

2838         \fi:
2839     \fi:
2840 \fi:
2841 #1
2842 \prg_do_nothing:
2843 #2
2844 \__cs_generate_variant_loop:nNwN { } #3 \q_mark
2845 }
2846 \cs_new:Npn \__cs_generate_variant_loop_base:N #1
2847 {
2848     \if:w c #1 N \else:
2849         \if:w o #1 n \else:
2850             \if:w V #1 n \else:
2851                 \if:w v #1 n \else:
2852                     \if:w f #1 n \else:
2853                         \if:w x #1 n \else:
2854                             \if:w n #1 n \else:
2855                                 \if:w N #1 N \else:
2856                                     \scan_stop:
2857                                 \fi:
2858                             \fi:
2859                         \fi:
2860                     \fi:
2861                 \fi:
2862             \fi:
2863         \fi:
2864     \fi:
2865 }
2866 \cs_new:Npn \__cs_generate_variant_loop_same:w
2867 #1 \prg_do_nothing: #2#3#4
2868 { #3 { #1 \__cs_generate_variant_same:N #2 } }
2869 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
2870 #1#2 \q_mark #3 ~ #4 \q_stop #5#6#7#8
2871 {
2872     \scan_stop: \scan_stop: \fi:
2873     \exp_not:N \q_mark
2874     \exp_not:N \q_stop
2875     \exp_not:N #6
2876     \exp_not:c { #7 : #8 #1 #3 }
2877 }
2878 \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \q_stop #2#3#4#5
2879 {
2880     \exp_not:n
2881     {
2882         \q_mark
2883         \__kernel_msg_error:nxx { kernel } { variant-too-long }
2884         {#5} { \token_to_str:N #3 }
2885         \use_none:nnn
2886         \q_stop
2887         #3
2888         #3
2889     }
2890 }
2891 \cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn

```

```

2892     #1#2 \fi: \fi: \fi: #3 \q_stop #4#5#6#7
2893   {
2894     \fi: \fi: \fi:
2895     \exp_not:n
2896     {
2897       \q_mark
2898       \__kernel_msg_error:nnxxxx { kernel } { invalid-variant }
2899       {#7} { \token_to_str:N #5 } {#1} {#2}
2900       \use_none:nnn
2901       \q_stop
2902       #5
2903       #5
2904     }
2905   }
2906 \cs_new:Npn \__cs_generate_variant_loop_special:NNwNNnn
2907   #1#2#3 \q_stop #4#5#6#7
2908   {
2909     #3 \q_stop #4 #5 {#6} {#7}
2910     \exp_not:n
2911     {
2912       \__cs_generate_variant_loop_warning:nnxxxx
2913       { kernel } { deprecated-variant }
2914       {#7} { \token_to_str:N #5 } {#1} {#2}
2915     }
2916   }
2917 \cs_new_protected:Npn \__cs_generate_variant_loop_warning:nnxxxx
2918   { \__kernel_msg_warning:nnxxxx }

```

(End definition for __cs_generate_variant_loop:nNwN and others.)

__cs_generate_variant_same:N

When the base and variant letters are identical, don't do any expansion. For most argument types, we can use the `n`-type no-expansion, but the `N` and `p` types require a slightly different behaviour with respect to braces. For `V`-type this function could output `N` to avoid adding useless braces but that is not a problem.

```

2919 \cs_new:Npn \__cs_generate_variant_same:N #1
2920   {
2921     \if:w N #1 N \else:
2922       \if:w p #1 p \else:
2923         n
2924       \if:w n #1 \else:
2925         \__cs_generate_variant_loop_special:NNwNNnn #1#1
2926       \fi:
2927     \fi:
2928   \fi:
2929   }

```

(End definition for __cs_generate_variant_same:N.)

__cs_generate_variant:wwNN

If the variant form has already been defined, log its existence (provided `log-functions` is active). Otherwise, make sure that the `\exp_args:N #3` form is defined, and if it contains `x`, change `__cs_tmp:w` locally to `\cs_new_protected:Npx`. Then define the variant by combining the `\exp_args:N #3` variant and the base function.

```

2930 \__kernel_patch:nnNNpn
2931   {

```

```

2932 \cs_if_free:NF #4
2933 {
2934   \__kernel_debug_log:x
2935   {
2936     Variant~\token_to_str:N #4~%
2937     already~defined;~ not~ changing~ it~ \msg_line_context:
2938   }
2939 }
2940 }
2941 { }
2942 \cs_new_protected:Npn \__cs_generate_variant:wwNN
2943 #1 \q_mark #2 \q_stop #3#4
2944 {
2945   #2
2946   \cs_if_free:NT #4
2947   {
2948     \group_begin:
2949     \__cs_generate_internal_variant:n {#1}
2950     \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
2951     \group_end:
2952   }
2953 }

```

(End definition for __cs_generate_variant:wwNN.)

```

\__cs_generate_internal_variant:n
\__cs_generate_internal_variant:wwnw
\__cs_generate_internal_variant_loop:n

```

Test if `\exp_args:N #1` is already defined and if not define it via the `\::` commands using the chars in #1. If #1 contains an x (this is the place where having converted the original comma-list argument to a string is very important), the result should be protected, and the next variant to be defined using that internal variant should be protected.

```

2954 \cs_new_protected:Npx \__cs_generate_internal_variant:n #1
2955 {
2956   \exp_not:N \__cs_generate_internal_variant:wwnNwnn
2957   #1 \exp_not:N \q_mark
2958   { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx }
2959   \cs_new_protected:cpx
2960   \token_to_str:N x \exp_not:N \q_mark
2961   { }
2962   \cs_new:cpx
2963   \exp_not:N \q_stop
2964   { exp_args:N #1 }
2965   {
2966     \exp_not:N \__cs_generate_internal_variant_loop:n #1
2967     { : \exp_not:N \use_i:nn }
2968   }
2969 }
2970 \exp_last_unbraced:NNNNo
2971 \cs_new_protected:Npn \__cs_generate_internal_variant:wwnNwnn #1
2972 { \token_to_str:N x } #2 \q_mark #3#4#5 \q_stop #6#7
2973 {
2974   #3
2975   \cs_if_free:cT {#6} { #4 {#6} {#7} }
2976 }

```

This command grabs char by char outputting `\::#1` (not expanded further). We avoid tests by putting a trailing `: \use_i:nn`, which leaves `\cs_end:` and removes the looping

macro. The colon is in fact also turned into \::: so that the required structure for \exp_args:N... commands is correctly terminated.

```

2977 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
2978 {
2979     \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
2980     \__cs_generate_internal_variant_loop:n
2981 }

```

(End definition for __cs_generate_internal_variant:n, __cs_generate_internal_variant:www, and __cs_generate_internal_variant_loop:n.)

```

\prg_generate_conditional_variant:Nnn
\__cs_generate_variant:nnNnn
\__cs_generate_variant:w
\__cs_generate_variant:n
\__cs_generate_variant_p_form:nnn
\__cs_generate_variant_T_form:nnn
\__cs_generate_variant_F_form:nnn
\__cs_generate_variant_TF_form:nnn

2982 \cs_new_protected:Npn \prg_generate_conditional_variant:Nnn #1
2983 {
2984     \use:x
2985     {
2986         \__cs_generate_variant:nnNnn
2987         \cs_split_function:N #1
2988     }
2989 }
2990 \cs_new_protected:Npn \__cs_generate_variant:nnNnn #1#2#3#4#5
2991 {
2992     \if_meaning:w \c_false_bool #3
2993     \__kernel_msg_error:nxx { kernel } { missing-colon }
2994     { \token_to_str:c {#1} }
2995     \use_i_delimit_by_q_stop:nw
2996     \fi:
2997     \exp_after:wN \__cs_generate_variant:w
2998     \tl_to_str:n {#5} , \scan_stop: , \q_recursion_stop
2999     \use_none_delimit_by_q_stop:w \q_mark {#1} {#2} {#4} \q_stop
3000 }
3001 \cs_new_protected:Npn \__cs_generate_variant:w
3002 #1 , #2 \q_mark #3#4#5
3003 {
3004     \if_meaning:w \scan_stop: #1 \scan_stop:
3005     \if_meaning:w \q_nil #1 \q_nil
3006     \use_i:nnn
3007     \fi:
3008     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
3009     \else:
3010     \cs_if_exist_use:cTF { __cs_generate_variant_#1_form:nnn }
3011     { {#3} {#4} {#5} }
3012     {
3013         \__kernel_msg_error:nxxx
3014         { kernel } { conditional-form-unknown }
3015         {#1} { \token_to_str:c { #3 : #4 } }
3016     }
3017     \fi:
3018     \__cs_generate_variant:w #2 \q_mark {#3} {#4} {#5}
3019 }
3020 \cs_new_protected:Npn \__cs_generate_variant_p_form:nnn #1#2
3021 { \cs_generate_variant:cn { #1 _p : #2 } }
3022 \cs_new_protected:Npn \__cs_generate_variant_T_form:nnn #1#2
3023 { \cs_generate_variant:cn { #1 : #2 T } }

```

```

3024 \cs_new_protected:Npn \__cs_generate_variant_F_form:nnn #1#2
3025   { \cs_generate_variant:cn { #1 : #2 F } }
3026 \cs_new_protected:Npn \__cs_generate_variant_TF_form:nnn #1#2
3027   { \cs_generate_variant:cn { #1 : #2 TF } }

```

(End definition for `\prg_generate_conditional_variant:Nnn` and others. This function is documented on page 237.)

```

\exp_args_generate:n
\__cs_args_generate:Nn
\__cs_args_generate:n

```

This function is not used in the kernel hence we can use functions that are defined in later modules. It also does not need to be fast so use inline mappings. For each requested variant we check that there are no characters besides `NnpcofVvx`, in particular that there are no spaces. Then we loop through the variant specifier and convert each letter to `\::<variant letter>`, with a trailing `\:::`.

```

3028 \cs_new_protected:Npn \exp_args_generate:n #1
3029   {
3030     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
3031     {
3032       \str_map_inline:nn {##1}
3033       {
3034         \str_if_in:nnF { NnpcofVvx } {####1}
3035         {
3036           \__kernel_msg_error:nnnn { kernel } { invalid-exp-args }
3037           {####1} {##1}
3038           \str_map_break:n { \use_none:nnnn }
3039         }
3040       }
3041       \exp_args:Nc \__cs_args_generate:Nn { exp_args:N ##1 } {##1}
3042     }
3043   }
3044 \cs_new_protected:Npn \__cs_args_generate:Nn #1#2
3045   {
3046     \cs_if_exist:NF #1
3047     {
3048       \str_if_in:nnTF {#2} { x } { \cs_new_protected:Npx } { \cs_new:Npx }
3049       #1 { \tl_map_function:nN { #2 : } \__cs_args_generate:n }
3050     }
3051   }
3052 \cs_new:Npn \__cs_args_generate:n #1 { \exp_not:c { :: #1 } }

```

(End definition for `\exp_args_generate:n`, `__cs_args_generate:Nn`, and `__cs_args_generate:n`. This function is documented on page 237.)

```

3053 </initex | package>

```

6 l3tl implementation

```

3054 <*initex | package>
3055 <@@=tl>

```

A token list variable is a $\text{T}_{\text{E}}\text{X}$ macro that holds tokens. By using the $\varepsilon\text{-T}_{\text{E}}\text{X}$ primitive `\unexpanded` inside a $\text{T}_{\text{E}}\text{X}$ `\edef` it is possible to store any tokens, including `#`, in this

way.

6.1 Functions

\tl_new:N Creating new token list variables is a case of checking for an existing definition and doing
\tl_new:c the definition.

```
3056 \cs_new_protected:Npn \tl_new:N #1
3057 {
3058     \__kernel_chk_if_free_cs:N #1
3059     \cs_gset_eq:NN #1 \c_empty_tl
3060 }
3061 \cs_generate_variant:Nn \tl_new:N { c }
```

(End definition for \tl_new:N. This function is documented on page 34.)

\tl_const:Nn Constants are also easy to generate.

```
\tl_const:Nx 3062 \__kernel_patch:nnNnpn { \__kernel_chk_var_scope:NN c #1 } { }
\tl_const:cn 3063 \cs_new_protected:Npn \tl_const:Nn #1#2
\tl_const:cx 3064 {
3065     \__kernel_chk_if_free_cs:N #1
3066     \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
3067 }
3068 \__kernel_patch:nnNnpn { \__kernel_chk_var_scope:NN c #1 } { }
3069 \cs_new_protected:Npn \tl_const:Nx #1#2
3070 {
3071     \__kernel_chk_if_free_cs:N #1
3072     \cs_gset_nopar:Npx #1 {#2}
3073 }
3074 \cs_generate_variant:Nn \tl_const:Nn { c }
3075 \cs_generate_variant:Nn \tl_const:Nx { c }
```

(End definition for \tl_const:Nn. This function is documented on page 34.)

\tl_clear:N Clearing a token list variable means setting it to an empty value. Error checking is sorted
\tl_clear:c out by the parent function.

```
\tl_gclear:N 3076 \cs_new_protected:Npn \tl_clear:N #1
\tl_gclear:c 3077 { \tl_set_eq:NN #1 \c_empty_tl }
3078 \cs_new_protected:Npn \tl_gclear:N #1
3079 { \tl_gset_eq:NN #1 \c_empty_tl }
3080 \cs_generate_variant:Nn \tl_clear:N { c }
3081 \cs_generate_variant:Nn \tl_gclear:N { c }
```

(End definition for \tl_clear:N and \tl_gclear:N. These functions are documented on page 34.)

\tl_clear_new:N Clearing a token list variable means setting it to an empty value. Error checking is sorted
\tl_clear_new:c out by the parent function.

```
\tl_gclear_new:N 3082 \cs_new_protected:Npn \tl_clear_new:N #1
\tl_gclear_new:c 3083 { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
3084 \cs_new_protected:Npn \tl_gclear_new:N #1
3085 { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
3086 \cs_generate_variant:Nn \tl_clear_new:N { c }
3087 \cs_generate_variant:Nn \tl_gclear_new:N { c }
```

(End definition for \tl_clear_new:N and \tl_gclear_new:N. These functions are documented on page 35.)

\tl_set_eq:NN For setting token list variables equal to each other. When checking is turned on, make sure both variables exist.

```

\tl_set_eq:Nc
\tl_set_eq:cN
\tl_set_eq:cc
3088 \__kernel_if_debug:TF
3089 {
3090   \cs_new_protected:Npn \tl_set_eq:NN #1#2
3091   {
3092     \__kernel_chk_var_local:N #1
3093     \__kernel_chk_var_exist:N #2
3094     \cs_set_eq:NN #1 #2
3095   }
3096   \cs_new_protected:Npn \tl_gset_eq:NN #1#2
3097   {
3098     \__kernel_chk_var_global:N #1
3099     \__kernel_chk_var_exist:N #2
3100     \cs_gset_eq:NN #1 #2
3101   }
3102 }
3103 {
3104   \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
3105   \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
3106 }
3107 \cs_generate_variant:Nn \tl_set_eq:NN { cN, Nc, cc }
3108 \cs_generate_variant:Nn \tl_gset_eq:NN { cN, Nc, cc }

```

(End definition for \tl_set_eq:NN and \tl_gset_eq:NN. These functions are documented on page 35.)

\tl_concat:NNN Concatenating token lists is easy. When checking is turned on, all three arguments must be checked: a token list #2 or #3 equal to \scan_stop: would lead to problems later on.

```

\tl_concat:ccc
\tl_gconcat:NNN
\tl_gconcat:ccc
3109 \__kernel_patch:nnNNpn
3110 {
3111   \__kernel_chk_var_exist:N #2
3112   \__kernel_chk_var_exist:N #3
3113 }
3114 { }
3115 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
3116 { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
3117 \__kernel_patch:nnNNpn
3118 {
3119   \__kernel_chk_var_exist:N #2
3120   \__kernel_chk_var_exist:N #3
3121 }
3122 { }
3123 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
3124 { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
3125 \cs_generate_variant:Nn \tl_concat:NNN { ccc }
3126 \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }

```

(End definition for \tl_concat:NNN and \tl_gconcat:NNN. These functions are documented on page 35.)

\tl_if_exist:p:N Copies of the cs functions defined in l3basics.

```

\tl_if_exist:p:c
\tl_if_exist:NTF
\tl_if_exist:cTF
3127 \prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
3128 \prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }

```

(End definition for \tl_if_exist:NTF. This function is documented on page 35.)

6.2 Constant token lists

\c_empty_tl Never full. We need to define that constant before using `\tl_new:N`.

```
3129 \tl_const:Nn \c_empty_tl { }
```

(End definition for `\c_empty_tl`. This variable is documented on page 46.)

\c_novalue_tl A special marker: as we don't have `\char_generate:nn` yet, has to be created the old-fashioned way.

```
3130 \group_begin:
3131 \tex_lccode:D 'A = '-
3132 \tex_lccode:D 'N = 'N
3133 \tex_lccode:D 'V = 'V
3134 \tex_lowercase:D
3135 {
3136   \group_end:
3137   \tl_const:Nn \c_novalue_tl { ANoValue- }
3138 }
```

(End definition for `\c_novalue_tl`. This variable is documented on page 47.)

\c_space_tl A space as a token list (as opposed to as a character).

```
3139 \tl_const:Nn \c_space_tl { ~ }
```

(End definition for `\c_space_tl`. This variable is documented on page 47.)

6.3 Adding to token list variables

\tl_set:Nn By using `\exp_not:n` token list variables can contain # tokens, which makes the token list registers provided by T_EX more or less redundant. The `\tl_set:No` version is done “by hand” as it is used quite a lot. Each definition is prefixed by a call to `__kernel_patch:nnNNpn` which adds an existence check to the definition.

```
\tl_set:Nf 3140 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\tl_set:Nx 3141 \cs_new_protected:Npn \tl_set:Nn #1#2
\tl_set:cn 3142 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
\tl_set:cV 3143 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\tl_set:cv 3144 \cs_new_protected:Npn \tl_set:No #1#2
\tl_set:co 3145 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
\tl_set:cf 3146 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\tl_set:cx 3147 \cs_new_protected:Npn \tl_set:Nx #1#2
3148 { \cs_set_nopar:Npx #1 {#2} }
\tl_gset:Nn 3149 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
\tl_gset:NV 3150 \cs_new_protected:Npn \tl_gset:Nn #1#2
\tl_gset:Nv 3151 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
\tl_gset:No 3152 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
\tl_gset:Nf 3153 \cs_new_protected:Npn \tl_gset:No #1#2
\tl_gset:Nx 3154 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
\tl_gset:cn 3155 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
\tl_gset:cV 3156 \cs_new_protected:Npn \tl_gset:Nx #1#2
\tl_gset:cv 3157 { \cs_gset_nopar:Npx #1 {#2} }
\tl_gset:co 3158 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
\tl_gset:cf 3159 \cs_generate_variant:Nn \tl_set:Nx { c }
\tl_gset:cx 3160 \cs_generate_variant:Nn \tl_set:Nn { c , co , cV , cv , cf }
3161 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
```

```

3162 \cs_generate_variant:Nn \tl_gset:Nx { c }
3163 \cs_generate_variant:Nn \tl_gset:Nn { c, co , cV , cv , cf }

```

(End definition for `\tl_set:Nn` and `\tl_gset:Nn`. These functions are documented on page 35.)

```

\tl_put_left:Nn Adding to the left is done directly to gain a little performance.
\tl_put_left:NV 3164 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\tl_put_left:No 3165 \cs_new_protected:Npn \tl_put_left:Nn #1#2
\tl_put_left:Nx 3166 { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\tl_put_left:cn 3167 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\tl_put_left:cV 3168 \cs_new_protected:Npn \tl_put_left:NV #1#2
\tl_put_left:co 3169 { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
\tl_put_left:cx 3170 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\tl_gput_left:Nn 3171 \cs_new_protected:Npn \tl_gput_left:No #1#2
\tl_gput_left:NV 3172 { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
\tl_gput_left:No 3173 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\tl_gput_left:Nx 3174 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
\tl_gput_left:cn 3175 { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
\tl_gput_left:cV 3176 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
\tl_gput_left:co 3177 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
\tl_gput_left:cx 3178 { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
3179 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
3180 \cs_new_protected:Npn \tl_gput_left:NV #1#2
3181 { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
3182 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
3183 \cs_new_protected:Npn \tl_gput_left:No #1#2
3184 { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
3185 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
3186 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
3187 { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
3188 \cs_generate_variant:Nn \tl_put_left:Nn { c }
3189 \cs_generate_variant:Nn \tl_put_left:NV { c }
3190 \cs_generate_variant:Nn \tl_put_left:No { c }
3191 \cs_generate_variant:Nn \tl_put_left:Nx { c }
3192 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
3193 \cs_generate_variant:Nn \tl_gput_left:NV { c }
3194 \cs_generate_variant:Nn \tl_gput_left:No { c }
3195 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

(End definition for `\tl_put_left:Nn` and `\tl_gput_left:Nn`. These functions are documented on page 35.)

```

\tl_put_right:Nn The same on the right.
\tl_put_right:NV 3196 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\tl_put_right:No 3197 \cs_new_protected:Npn \tl_put_right:Nn #1#2
\tl_put_right:Nx 3198 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\tl_put_right:cn 3199 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\tl_put_right:cV 3200 \cs_new_protected:Npn \tl_put_right:NV #1#2
\tl_put_right:co 3201 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\tl_put_right:cx 3202 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\tl_gput_right:Nn 3203 \cs_new_protected:Npn \tl_gput_right:No #1#2
\tl_gput_right:NV 3204 { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\tl_gput_right:No 3205 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\tl_gput_right:Nx 3206 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
\tl_gput_right:cn 3207 { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
\tl_gput_right:cV
\tl_gput_right:co
\tl_gput_right:cx

```

```

3208 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
3209 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
3210   { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
3211 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
3212 \cs_new_protected:Npn \tl_gput_right:NV #1#2
3213   { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
3214 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
3215 \cs_new_protected:Npn \tl_gput_right:No #1#2
3216   { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
3217 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
3218 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
3219   { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
3220 \cs_generate_variant:Nn \tl_put_right:Nn { c }
3221 \cs_generate_variant:Nn \tl_put_right:NV { c }
3222 \cs_generate_variant:Nn \tl_put_right:No { c }
3223 \cs_generate_variant:Nn \tl_put_right:Nx { c }
3224 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
3225 \cs_generate_variant:Nn \tl_gput_right:NV { c }
3226 \cs_generate_variant:Nn \tl_gput_right:No { c }
3227 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

(End definition for `\tl_put_right:Nn` and `\tl_gput_right:Nn`. These functions are documented on page 35.)

6.4 Reassigning token list category codes

`\c__tl_rescan_marker_tl`

The rescanning code needs a special token list containing the same character (chosen here to be a colon) with two different category codes: it cannot appear in the tokens being rescanned since all colons have the same category code.

```

3228 \tl_const:Nx \c__tl_rescan_marker_tl { : \token_to_str:N : }

```

(End definition for `\c__tl_rescan_marker_tl`.)

`\tl_set_rescan:Nnn`

`\tl_set_rescan:Nno`

`\tl_set_rescan:Nnx`

`\tl_set_rescan:cnn`

`\tl_set_rescan:cno`

`\tl_set_rescan:cnx`

`\tl_gset_rescan:Nnn`

`\tl_gset_rescan:Nno`

`\tl_gset_rescan:Nnx`

`\tl_gset_rescan:cnn`

`\tl_gset_rescan:cno`

`\tl_gset_rescan:cnx`

`\tl_rescan:nn`

`__tl_set_rescan:NNnn`

`__tl_set_rescan_multi:n`

`__tl_rescan:w`

These functions use a common auxiliary. After some initial setup explained below, and the user setup #3 (followed by `\scan_stop:` to be safe), the tokens are rescanned by `__tl_set_rescan:n` and stored into `\l__tl_internal_a_tl`, then passed to #1#2 outside the group after expansion. The auxiliary `__tl_set_rescan:n` is defined later: in the simplest case, this auxiliary calls `__tl_set_rescan_multi:n`, whose code is included here to help understand the approach.

One difficulty when rescanning is that `\scantokens` treats the argument as a file, and without the correct settings a T_EX error occurs:

```
! File ended while scanning definition of ...
```

The standard solution is to use an x-expanding assignment and set `\everyeof` to `\exp_not:N` to suppress the error at the end of the file. Since the rescanned tokens should not be expanded, they are taken as a delimited argument of an auxiliary which wraps them in `\exp_not:n` (in fact `\exp_not:o`, as there is a `\prg_do_nothing:` to avoid losing braces). The delimiter cannot appear within the rescanned token list because it contains twice the same character, with different catcodes.

The difference between single-line and multiple-line files complicates the story, as explained below.

```

3229 \cs_new_protected:Npn \tl_set_rescan:Nnn

```

```

3230 { \_tl\_set\_rescan:NNnn \tl\_set:Nn }
3231 \cs\_new\_protected:Npn \tl\_gset\_rescan:Nnn
3232 { \_tl\_set\_rescan:NNnn \tl\_gset:Nn }
3233 \cs\_new\_protected:Npn \tl\_rescan:nn
3234 { \_tl\_set\_rescan:NNnn \prg\_do\_nothing: \use:n }
3235 \cs\_new\_protected:Npn \_tl\_set\_rescan:NNnn #1#2#3#4
3236 {
3237   \tl\_if\_empty:nTF {#4}
3238   {
3239     \group\_begin:
3240     #3
3241     \group\_end:
3242     #1 #2 { }
3243   }
3244   {
3245     \group\_begin:
3246     \exp\_args:No \etex\_everyeof:D
3247     { \c\_tl\_rescan\_marker\_tl \exp\_not:N }
3248     \int\_compare:nNnT \tex\_endlinechar:D = { 32 }
3249     { \int\_set:Nn \tex\_endlinechar:D { -1 } }
3250     \tex\_newlinechar:D \tex\_endlinechar:D
3251     #3 \scan\_stop:
3252     \exp\_args:No \_tl\_set\_rescan:n { \tl\_to\_str:n {#4} }
3253     \exp\_args:NNNo
3254     \group\_end:
3255     #1 #2 \l\_tl\_internal\_a\_tl
3256   }
3257 }
3258 \cs\_new\_protected:Npn \_tl\_set\_rescan\_multi:n #1
3259 {
3260   \tl\_set:Nx \l\_tl\_internal\_a\_tl
3261   {
3262     \exp\_after:wN \_tl\_rescan:w
3263     \exp\_after:wN \prg\_do\_nothing:
3264     \etex\_scantokens:D {#1}
3265   }
3266 }
3267 \exp\_args:Nno \use:nn
3268 { \cs\_new:Npn \_tl\_rescan:w #1 } \c\_tl\_rescan\_marker\_tl
3269 { \exp\_not:o {#1} }
3270 \cs\_generate\_variant:Nn \tl\_set\_rescan:Nnn { Nno , Nnx }
3271 \cs\_generate\_variant:Nn \tl\_set\_rescan:Nnn { c , cno , cnx }
3272 \cs\_generate\_variant:Nn \tl\_gset\_rescan:Nnn { Nno , Nnx }
3273 \cs\_generate\_variant:Nn \tl\_gset\_rescan:Nnn { c , cno }

```

(End definition for `\tl_set_rescan:Nnn` and others. These functions are documented on page 37.)

<pre> _tl_set_rescan:n _tl_set_rescan:NnTF _tl_set_rescan_single:nn _tl_set_rescan_single_aux:nn </pre>	<p>This function calls <code>_tl_set_rescan_multiple:n</code> or <code>_tl_set_rescan_single:nn</code> { ' } depending on whether its argument is a single-line fragment of code/data or is made of multiple lines by testing for the presence of a <code>\newlinechar</code> character. If <code>\newlinechar</code> is out of range, the argument is assumed to be a single line.</p>
--	---

The case of multiple lines is a straightforward application of `\scantokens` as described above. The only subtlety is that `\newlinechar` should be equal to `\endlinechar` because `\newlinechar` characters become new lines and then become `\endlinechar`

characters when writing to an abstract file and reading back. This equality is ensured by setting `\newlinechar` equal to `\endlinechar`. Prior to this, `\endlinechar` is set to `-1` if it was `32` (in particular true after `\ExplSyntaxOn`) to avoid unreasonable line-breaks at every space for instance in error messages triggered by the user setup. Another side effect of reading back from the file is that spaces (catcode `10`) are ignored at the beginning of lines, and spaces and tabs (character code `32` and `9`) are ignored at the end of lines.

For a single line, no `\endlinechar` should be added, so it is set to `-1`, and spaces should not be removed.

Trailing spaces and tabs are a difficult matter, as `TEX` removes these at a very low level. The only way to preserve them is to rescan not the argument but the argument followed by a character with a reasonable category code. Here, `11` (letter), `12` (other) and `13` (active) are accepted, as these are suitable for delimiting an argument, and it is very unlikely that none of the ASCII characters are in one of these categories. To avoid selecting one particular character to put at the end, whose category code may have been modified, there is a loop through characters from `'` (ASCII `39`) to `~` (ASCII `127`). The choice of starting point was made because this is the start of a very long range of characters whose standard category is letter or other, thus minimizing the number of steps needed by the loop (most often just a single one). Once a valid character is found, run some code very similar to `__tl_set_rescan_multi:n`, except that `__tl_rescan:w` must be redefined to also remove the additional character (with the appropriate catcode). Getting the delimiter with the right catcode requires using `\scantokens` inside an `x`-expansion, hence using the previous definition of `__tl_rescan:w` as well. The odd `\exp_not:N` `\use:n` ensures that the trailing `\exp_not:N` in `\everyeof` does not prevent the expansion of `\c__tl_rescan_marker_tl`, but rather of a closing brace (this does nothing). If no valid character is found, similar code is ran, and the only difference is that trailing spaces are not preserved (bear in mind that this only happens if no character between `39` and `127` has catcode letter, other or active).

There is also some work to preserve leading spaces: test whether the first character (given by `\str_head:n`, with an extra space to circumvent a limitation of `f`-expansion) has catcode `10` and add what `TEX` would add in the middle of a line for any sequence of such characters: a single space with catcode `10` and character code `32`.

```

3274 \group_begin:
3275   \tex_catcode:D '\^~@ = 12 \scan_stop:
3276   \cs_new_protected:Npn \__tl_set_rescan:n #1
3277     {
3278       \int_compare:nNnTF \tex_newlinechar:D < 0
3279         { \use_ii:nn }
3280         {
3281           \char_set_lccode:nn { 0 } { \tex_newlinechar:D }
3282           \tex_lowercase:D { \__tl_set_rescan:NnTF ^~@ } {#1}
3283         }
3284         { \__tl_set_rescan_multi:n }
3285         { \__tl_set_rescan_single:nn { ' } }
3286       {#1}
3287     }
3288   \cs_new_protected:Npn \__tl_set_rescan:NnTF #1#2
3289     { \tl_if_in:nnTF {#2} {#1} }
3290   \cs_new_protected:Npn \__tl_set_rescan_single:nn #1
3291     {
3292       \int_compare:nNnTF
3293         { \char_value_catcode:n { '#1 } / 3 } = 4

```

```

3294 { \_tl\_set\_rescan\_single\_aux:nn {#1} }
3295 {
3296   \int\_compare:nNnTF { '#1 } < { '\~ }
3297   {
3298     \char\_set\_lccode:nn { 0 } { '#1 + 1 }
3299     \tex\_lowercase:D { \_tl\_set\_rescan\_single:nn { ^~@ } }
3300   }
3301   { \_tl\_set\_rescan\_single\_aux:nn { } }
3302 }
3303 }
3304 \cs\_new\_protected:Npn \_tl\_set\_rescan\_single\_aux:nn #1#2
3305 {
3306   \int\_set:Nn \tex\_endlinechar:D { -1 }
3307   \use:x
3308   {
3309     \exp\_not:N \use:n
3310     {
3311       \exp\_not:n { \cs\_set:Npn \_tl\_rescan:w ##1 }
3312       \exp\_after:wN \_tl\_rescan:w
3313       \exp\_after:wN \prg\_do\_nothing:
3314       \etex\_scantokens:D {#1}
3315     }
3316     \c\_tl\_rescan\_marker\_tl
3317   }
3318   { \exp\_not:o {##1} }
3319   \tl\_set:Nx \l\_tl\_internal\_a\_tl
3320   {
3321     \int\_compare:nNnT
3322     {
3323       \char\_value\_catcode:n
3324       { \exp\_last\_unbraced:Nf ' { \str\_head:n {#2} } ~ }
3325     }
3326     = { 10 } { ~ }
3327     \exp\_after:wN \_tl\_rescan:w
3328     \exp\_after:wN \prg\_do\_nothing:
3329     \etex\_scantokens:D { #2 #1 }
3330   }
3331 }
3332 \group\_end:

```

(End definition for `_tl_set_rescan:n` and others.)

6.5 Modifying token list variables

`\tl_replace_all:Nnn` All of the `replace` functions call `_tl_replace:NnNNNnn` with appropriate arguments.
`\tl_replace_all:cnn` The first two arguments are explained later. The next controls whether the replacement
`\tl_greplace_all:Nnn` function calls itself (`_tl_replace_next:w`) or stops (`_tl_replace_wrap:w`) after
`\tl_greplace_all:cnn` the first replacement. Next comes an x-type assignment function `\tl_set:Nx` or `\tl_-`
`\tl_replace_once:Nnn` `gset:Nx` for local or global replacements. Finally, the three arguments $\langle tl\ var \rangle$ $\{ \langle pattern \rangle \}$
`\tl_replace_once:cnn` $\{ \langle replacement \rangle \}$ provided by the user. When describing the auxiliary functions below,
`\tl_greplace_once:Nnn` we denote the contents of the $\langle tl\ var \rangle$ by $\langle token\ list \rangle$.
`\tl_greplace_once:cnn`

```

3333 \cs\_new\_protected:Npn \tl\_replace\_once:Nnn
3334 { \_tl\_replace:NnNNNnn \q\_mark ? \_tl\_replace\_wrap:w \tl\_set:Nx }

```



```

3335 \cs_new_protected:Npn \tl_greplace_once:Nnn
3336 { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_wrap:w \tl_gset:Nx }
3337 \cs_new_protected:Npn \tl_replace_all:Nnn
3338 { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_next:w \tl_set:Nx }
3339 \cs_new_protected:Npn \tl_greplace_all:Nnn
3340 { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_next:w \tl_gset:Nx }
3341 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
3342 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
3343 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
3344 \cs_generate_variant:Nn \tl_greplace_all:Nnn { c }

```

(End definition for `\tl_replace_all:Nnn` and others. These functions are documented on page 36.)

```

\__tl_replace:NnNNNnn
\__tl_replace_auxi:NnnNNNnn
\__tl_replace_auxii:NnnNNnn
  \__tl_replace_next:w
  \__tl_replace_wrap:w

```

To implement the actual replacement auxiliary `__tl_replace_auxii:NnnNNnn` we need a $\langle \textit{delimiter} \rangle$ with the following properties:

- all occurrences of the $\langle \textit{pattern} \rangle$ #6 in “ $\langle \textit{token list} \rangle \langle \textit{delimiter} \rangle$ ” belong to the $\langle \textit{token list} \rangle$ and have no overlap with the $\langle \textit{delimiter} \rangle$,
- the first occurrence of the $\langle \textit{delimiter} \rangle$ in “ $\langle \textit{token list} \rangle \langle \textit{delimiter} \rangle$ ” is the trailing $\langle \textit{delimiter} \rangle$.

We first find the building blocks for the $\langle \textit{delimiter} \rangle$, namely two tokens $\langle A \rangle$ and $\langle B \rangle$ such that $\langle A \rangle$ does not appear in #6 and #6 is not $\langle B \rangle$ (this condition is trivial if #6 has more than one token). Then we consider the delimiters “ $\langle A \rangle$ ” and “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ”, for $n \geq 1$, where $\langle A \rangle^n$ denotes n copies of $\langle A \rangle$, and we choose as our $\langle \textit{delimiter} \rangle$ the first one which is not in the $\langle \textit{token list} \rangle$.

Every delimiter in the set obeys the first condition: #6 does not contain $\langle A \rangle$ hence cannot be overlapping with the $\langle \textit{token list} \rangle$ and the $\langle \textit{delimiter} \rangle$, and it cannot be within the $\langle \textit{delimiter} \rangle$ since it would have to be in one of the two $\langle B \rangle$ hence be equal to this single token (or empty, but this is an error case filtered separately). Given the particular form of these delimiters, for which no prefix is also a suffix, the second condition is actually a consequence of the weaker condition that the $\langle \textit{delimiter} \rangle$ we choose does not appear in the $\langle \textit{token list} \rangle$. Additionally, the set of delimiters is such that a $\langle \textit{token list} \rangle$ of n tokens can contain at most $O(n^{1/2})$ of them, hence we find a $\langle \textit{delimiter} \rangle$ with at most $O(n^{1/2})$ tokens in a time at most $O(n^{3/2})$. Bear in mind that these upper bounds are reached only in very contrived scenarios: we include the case “ $\langle A \rangle$ ” in the list of delimiters to try, so that the $\langle \textit{delimiter} \rangle$ is simply `\q_mark` in the most common situation where neither the $\langle \textit{token list} \rangle$ nor the $\langle \textit{pattern} \rangle$ contains `\q_mark`.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty $\langle \textit{pattern} \rangle$ #6 is an error, and if #1 is absent from both the $\langle \textit{token list} \rangle$ #5 and the $\langle \textit{pattern} \rangle$ #6 then we can use it as the $\langle \textit{delimiter} \rangle$ through `__tl_replace_auxii:NnnNNnn {#1}`. Otherwise, we end up calling `__tl_replace:NnNNNnn` repeatedly with the first two arguments `\q_mark {?}`, `\? {??}`, `\?? {???}`, and so on, until #6 does not contain the control sequence #1, which we take as our $\langle A \rangle$. The argument #2 only serves to collect ? characters for #1. Note that the order of the tests means that the first two are done every time, which is wasteful (for instance, we repeatedly test for the emptiness of #6). However, this is rare enough not to matter. Finally, choose $\langle B \rangle$ to be `\q_nil` or `\q_stop` such that it is not equal to #6.

The `__tl_replace_auxi:NnnNNNnn` auxiliary receives $\{\langle A \rangle\}$ and $\{\langle A \rangle^n \langle B \rangle\}$ as its arguments, initially with $n = 1$. If “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ” is in the $\langle \textit{token list} \rangle$ then

increase n and try again. Once it is not anymore in the $\langle token\ list \rangle$ we take it as our $\langle delimiter \rangle$ and pass this to the `auxii` auxiliary.

```

3345 \cs_new_protected:Npn \__tl_replace:NnNNNnn #1#2#3#4#5#6#7
3346 {
3347   \tl_if_empty:nTF {#6}
3348   {
3349     \_kernel_msg_error:nxx { kernel } { empty-search-pattern }
3350     { \tl_to_str:n {#7} }
3351   }
3352   {
3353     \tl_if_in:ontF { #5 #6 } {#1}
3354     {
3355       \tl_if_in:nnTF {#6} {#1}
3356       { \exp_args:Nc \__tl_replace:NnNNNnn {#2} {#2?} }
3357       {
3358         \quark_if_nil:nTF {#6}
3359         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_stop } }
3360         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_nil } }
3361       }
3362     }
3363     { \__tl_replace_auxii:nNNNnn {#1} }
3364     #3#4#5 {#6} {#7}
3365   }
3366 }
3367 \cs_new_protected:Npn \__tl_replace_auxi:NnnNNNnn #1#2#3
3368 {
3369   \tl_if_in:NnTF #1 { #2 #3 #3 }
3370   { \__tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
3371   { \__tl_replace_auxii:nNNNnn { #2 #3 #3 } }
3372 }

```

The auxiliary `__tl_replace_auxii:nNNNnn` receives the following arguments:

$\{ \langle delimiter \rangle \}$ $\langle function \rangle$ $\langle assignment \rangle$
 $\langle tl\ var \rangle$ $\{ \langle pattern \rangle \}$ $\{ \langle replacement \rangle \}$

All of its work is done between `\group_align_safe_begin:` and `\group_align_safe_end:` to avoid issues in alignments. It does the actual replacement within `#3 #4 { ... }`, an x -expanding $\langle assignment \rangle$ `#3` to the $\langle tl\ var \rangle$ `#4`. The auxiliary `__tl_replace_next:w` is called, followed by the $\langle token\ list \rangle$, some tokens including the $\langle delimiter \rangle$ `#1`, followed by the $\langle pattern \rangle$ `#5`. This auxiliary finds an argument delimited by `#5` (the presence of a trailing `#5` avoids runaway arguments) and calls `__tl_replace_wrap:w` to test whether this `#5` is found within the $\langle token\ list \rangle$ or is the trailing one.

If on the one hand it is found within the $\langle token\ list \rangle$, then `##1` cannot contain the $\langle delimiter \rangle$ `#1` that we worked so hard to obtain, thus `__tl_replace_wrap:w` gets `##1` as its own argument `##1`, and protects it against the x -expanding assignment. It also finds `\exp_not:n` as `##2` and does nothing to it, thus letting through `\exp_not:n { \langle replacement \rangle }` into the assignment. Note that `__tl_replace_next:w` and `__tl_replace_wrap:w` are always called followed by two empty brace groups. These are safe because no delimiter can match them. They prevent losing braces when grabbing delimited arguments, but require the use of `\exp_not:o` and `\use_none:nn`, rather than simply `\exp_not:n`. Afterwards, `__tl_replace_next:w` is called to repeat the replacement, or `__tl_replace_wrap:w` if we only want a single replacement. In this second

case, ##1 is the *<remaining tokens>* in the *<token list>* and ##2 is some *<ending code>* which ends the assignment and removes the trailing tokens #5 using some `\if_false: { \fi: }` trickery because #5 may contain any delimiter.

If on the other hand the argument ##1 of `__tl_replace_next:w` is delimited by the trailing *<pattern>* #5, then ##1 is “{ } { } *<token list>* *<delimiter>* {*<ending code>*}”, hence `__tl_replace_wrap:w` finds “{ } { } *<token list>*” as ##1 and the *<ending code>* as ##2. It leaves the *<token list>* into the assignment and unbraces the *<ending code>* which removes what remains (essentially the *<delimiter>* and *<replacement>*).

```

3373 \cs_new_protected:Npn \__tl_replace_auxii:nNNNnn #1#2#3#4#5#6
3374 {
3375   \group_align_safe_begin:
3376   \cs_set:Npn \__tl_replace_wrap:w ##1 #1 ##2
3377     { \exp_not:o { \use_none:nn ##1 } ##2 }
3378   \cs_set:Npx \__tl_replace_next:w ##1 #5
3379   {
3380     \exp_not:N \__tl_replace_wrap:w ##1
3381     \exp_not:n { #1 }
3382     \exp_not:n { \exp_not:n {#6} }
3383     \exp_not:n { #2 { } { } }
3384   }
3385   #3 #4
3386   {
3387     \exp_after:wN \__tl_replace_next:w
3388     \exp_after:wN { \exp_after:wN }
3389     \exp_after:wN { \exp_after:wN }
3390     #4
3391     #1
3392     {
3393       \if_false: { \fi: }
3394       \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
3395     }
3396     #5
3397   }
3398   \group_align_safe_end:
3399 }
3400 \cs_new_eq:NN \__tl_replace_wrap:w ?
3401 \cs_new_eq:NN \__tl_replace_next:w ?

```

(End definition for `__tl_replace:NnNNNnn` and others.)

```

\tl_remove_once:Nn Removal is just a special case of replacement.
\tl_remove_once:cn 3402 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
\tl_gremove_once:Nn 3403 { \tl_replace_once:Nnn #1 {#2} { } }
\tl_gremove_once:cn 3404 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
3405 { \tl_greplace_once:Nnn #1 {#2} { } }
3406 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
3407 \cs_generate_variant:Nn \tl_gremove_once:Nn { c }

```

(End definition for `\tl_remove_once:Nn` and `\tl_gremove_once:Nn`. These functions are documented on page 36.)

```

\tl_remove_all:Nn Removal is just a special case of replacement.
\tl_remove_all:cn 3408 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
\tl_gremove_all:Nn 3409 { \tl_replace_all:Nnn #1 {#2} { } }
\tl_gremove_all:cn

```

```

3410 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
3411 { \tl_greplace_all:Nnn #1 {#2} { } }
3412 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
3413 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }

```

(End definition for `\tl_remove_all:Nn` and `\tl_gremove_all:Nn`. These functions are documented on page 36.)

6.6 Token list conditionals

`\tl_if_blank_p:n` TeX skips spaces when reading a non-delimited arguments. Thus, a *token list* is blank if and only if `\use_none:n <token list> ?` is empty after one expansion. The auxiliary `\tl_if_blank_p:V` `_tl_if_empty_return:o` is a fast emptiness test, converting its argument to a string `\tl_if_blank_p:o` (after one expansion) and using the test `\if_meaning:w \q_nil ... \q_nil`.
`\tl_if_blank:nTF` 3414 `\prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }`
`\tl_if_blank:VTF` 3415 `{ _tl_if_empty_return:o { \use_none:n #1 ? } }`
`\tl_if_blank:oTF` 3416 `\prg_generate_conditional_variant:Nnn \tl_if_blank:n`
`_tl_if_blank_p:NNw` 3417 `{ V , o } { p , T , F , TF }`

(End definition for `\tl_if_blank:nTF` and `_tl_if_blank_p:NNw`. This function is documented on page 37.)

`\tl_if_empty_p:N` These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).
`\tl_if_empty_p:c`
`\tl_if_empty:nTF` 3418 `\prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }`
`\tl_if_empty:cTF` 3419 `{`
3420 `\if_meaning:w #1 \c_empty_tl`
3421 `\prg_return_true:`
3422 `\else:`
3423 `\prg_return_false:`
3424 `\fi:`
3425 `}`
3426 `\prg_generate_conditional_variant:Nnn \tl_if_empty:N`
3427 `{ c } { p , T , F , TF }`

(End definition for `\tl_if_empty:nTF`. This function is documented on page 38.)

`\tl_if_empty_p:n` Convert the argument to a string: this is empty if and only if the argument is. Then
`\tl_if_empty_p:V` `\if_meaning:w \q_nil ... \q_nil` is true if and only if the string ... is empty. It
`\tl_if_empty:nTF` could be tempting to use `\if_meaning:w \q_nil #1 \q_nil` directly. This fails on a
`\tl_if_empty:VTF` token list starting with `\q_nil` of course but more troubling is the case where argument
is a complete conditional such as `\if_true: a \else: b \fi:` because then `\if_true:`
is used by `\if_meaning:w`, the test turns out false, the `\else:` executes the false
branch, the `\fi:` ends it and the `\q_nil` at the end starts executing...

```

3428 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
3429 {
3430   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
3431   \tl_to_str:n {#1} \q_nil
3432   \prg_return_true:
3433   \else:
3434   \prg_return_false:
3435   \fi:
3436 }
3437 \prg_generate_conditional_variant:Nnn \tl_if_empty:n
3438 { V } { p , TF , T , F }

```

(End definition for \tl_if_empty:nTF. This function is documented on page 38.)

\tl_if_empty_p:o The auxiliary function __tl_if_empty_return:o is for use in various token list conditionals which reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on \tl_if_empty:nTF, but the expansion is hard-coded for efficiency, as this auxiliary function is used in many places.

```

3439 \cs_new:Npn \__tl_if_empty_return:o #1
3440 {
3441   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
3442   \__kernel_tl_to_str:w \exp_after:wN {#1} \q_nil
3443   \prg_return_true:
3444   \else:
3445     \prg_return_false:
3446   \fi:
3447 }
3448 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
3449 { \__tl_if_empty_return:o {#1} }

```

(End definition for \tl_if_empty:nTF and __tl_if_empty_return:o. This function is documented on page 38.)

\tl_if_eq_p:NN Returns \c_true_bool if and only if the two token list variables are equal.

```

\tl_if_eq_p:Nc 3450 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
\tl_if_eq_p:cN 3451 {
\tl_if_eq_p:cc 3452   \if_meaning:w #1 #2
\tl_if_eq:NNTF 3453   \prg_return_true:
\tl_if_eq:NcTF 3454   \else:
\tl_if_eq:cNTF 3455   \prg_return_false:
\tl_if_eq:ccTF 3456   \fi:
3457 }
3458 \prg_generate_conditional_variant:Nnn \tl_if_eq:NN
3459 { Nc , c , cc } { p , TF , T , F }

```

(End definition for \tl_if_eq:NNTF. This function is documented on page 38.)

\tl_if_eq:nnTF A simple store and compare routine.

```

\l__tl_internal_a_tl 3460 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
\l__tl_internal_b_tl 3461 {
3462   \group_begin:
3463     \tl_set:Nn \l__tl_internal_a_tl {#1}
3464     \tl_set:Nn \l__tl_internal_b_tl {#2}
3465     \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
3466     \group_end:
3467     \prg_return_true:
3468     \else:
3469       \group_end:
3470       \prg_return_false:
3471     \fi:
3472   }
3473   \tl_new:N \l__tl_internal_a_tl
3474   \tl_new:N \l__tl_internal_b_tl

```

(End definition for \tl_if_eq:nnTF, \l__tl_internal_a_tl, and \l__tl_internal_b_tl. This function is documented on page 38.)

`\tl_if_in:NnTF` See `\tl_if_in:nnTF` for further comments. Here we simply expand the token list variable `\tl_if_in:cnTF` and pass it to `\tl_if_in:nnTF`.

```

3475 \cs_new_protected:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
3476 \cs_new_protected:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
3477 \cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
3478 \prg_generate_conditional_variant:Nnn \tl_if_in:Nn
3479 { c } { T , F , TF }

```

(End definition for `\tl_if_in:NnTF`. This function is documented on page 38.)

`\tl_if_in:nnTF` Once more, the test relies on the emptiness test for robustness. The function `__tl_tmp:w` removes tokens until the first occurrence of `#2`. If this does not appear in `#1`, then the final `#2` is removed, leaving an empty token list. Otherwise some tokens remain, and the test is false. See `\tl_if_empty:nTF` for details on the emptiness test.

Treating correctly cases like `\tl_if_in:nnTF {a state}{states}`, where `#1#2` contains `#2` before the end, requires special care. To cater for this case, we insert `{ }{ }` between the two token lists. This marker may not appear in `#2` because of \TeX limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments. The `\if_false:` constructions are a faster way to do `\group_align_safe_begin:` and `\group_align_safe_end:`.

```

3480 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
3481 {
3482   \if_false: { \fi:
3483     \cs_set:Npn \__tl_tmp:w ##1 #2 { }
3484     \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
3485       { \prg_return_false: } { \prg_return_true: }
3486     \if_false: } \fi:
3487   }
3488   \prg_generate_conditional_variant:Nnn \tl_if_in:nn
3489   { V , o , no } { T , F , TF }

```

(End definition for `\tl_if_in:nnTF`. This function is documented on page 38.)

`\tl_if_novalue_p:n` Tests for `-NoValue-`: this is similar to `\tl_if_in:nn` but set up to be expandable and `\tl_if_novalue:nTF` to check the value exactly. The question mark prevents the auxiliary from losing braces.

```

\__tl_if_novalue:w
3490 \cs_set_protected:Npn \__tl_tmp:w #1
3491 {
3492   \prg_new_conditional:Npnn \tl_if_novalue:n ##1
3493   { p , T , F , TF }
3494   {
3495     \str_if_eq:onTF
3496       { \__tl_if_novalue:w ? ##1 { } #1 }
3497       { ? { } #1 }
3498       { \prg_return_true: }
3499       { \prg_return_false: }
3500   }
3501   \cs_new:Npn \__tl_if_novalue:w ##1 #1 {##1}
3502 }
3503 \exp_args:No \__tl_tmp:w { \c_novalue_tl }

```

(End definition for `\tl_if_novalue:nTF` and `__tl_if_novalue:w`. This function is documented on page 38.)

`\tl_if_single_p:N` Expand the token list and feed it to `\tl_if_single:n`.
`\tl_if_single:NTF`

```

3504 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
3505 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
3506 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }
3507 \cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }

```

(End definition for `\tl_if_single:NTF`. This function is documented on page 39.)

`\tl_if_single_p:n` This test is similar to `\tl_if_empty:nTF`. Expanding `\use_none:nn #1 ??` once yields
`\tl_if_single:nTF` an empty result if #1 is blank, a single ? if #1 has a single item, and otherwise yields some
`__tl_if_single_p:n` tokens ending with ??. Then, `\tl_to_str:n` makes sure there are no odd category codes.
`__tl_if_single:nTF` An earlier version would compare the result to a single ? using string comparison, but
the Lua call is slow in LuaTeX. Instead, `__tl_if_single:nnw` picks the second token
in front of it. If #1 is empty, this token is the trailing ? and the catcode test yields `false`.
If #1 has a single item, the token is ^ and the catcode test yields `true`. Otherwise, it is
one of the characters resulting from `\tl_to_str:n`, and the catcode test yields `false`.
Note that `\if_catcode:w` and `__kernel_tl_to_str:w` are primitives that take care of
expansion.

```

3508 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
3509 {
3510   \if_catcode:w ^ \exp_after:wN \__tl_if_single:nnw
3511     \__kernel_tl_to_str:w
3512     \exp_after:wN { \use_none:nn #1 ?? } ^ ? \q_stop
3513   \prg_return_true:
3514   \else:
3515     \prg_return_false:
3516   \fi:
3517 }
3518 \cs_new:Npn \__tl_if_single:nnw #1#2#3 \q_stop {#2}

```

(End definition for `\tl_if_single:nTF` and `__tl_if_single:nTF`. This function is documented on page 39.)

`\tl_case:Nn` The aim here is to allow the case statement to be evaluated using a known number of
`\tl_case:cn` expansion steps (two), and without needing to use an explicit “end of recursion” marker.
`\tl_case:NnTF` That is achieved by using the test input as the final case, as this is always true. The
`\tl_case:cnTF` trick is then to tidy up the output such that the appropriate case code plus either the
`__tl_case:nnTF` true or false branch code is inserted.

```

\__tl_case:Nw
\__tl_case_end:nw
3519 \cs_new:Npn \tl_case:Nn #1#2
3520 {
3521   \exp:w
3522   \__tl_case:NnTF #1 {#2} { } { }
3523 }
3524 \cs_new:Npn \tl_case:NnT #1#2#3
3525 {
3526   \exp:w
3527   \__tl_case:NnTF #1 {#2} {#3} { }
3528 }
3529 \cs_new:Npn \tl_case:NnF #1#2#3
3530 {
3531   \exp:w
3532   \__tl_case:NnTF #1 {#2} { } {#3}
3533 }

```

```

3534 \cs_new:Npn \tl_case:NnTF #1#2
3535 {
3536   \exp:w
3537   \__tl_case:NnTF #1 {#2}
3538 }
3539 \cs_new:Npn \__tl_case:NnTF #1#2#3#4
3540 { \__tl_case:Nw #1 #2 #1 { } \q_mark {#3} \q_mark {#4} \q_stop }
3541 \cs_new:Npn \__tl_case:Nw #1#2#3
3542 {
3543   \tl_if_eq:NNTF #1 #2
3544   { \__tl_case_end:nw {#3} }
3545   { \__tl_case:Nw #1 }
3546 }
3547 \cs_generate_variant:Nn \tl_case:Nn { c }
3548 \prg_generate_conditional_variant:Nnn \tl_case:Nn
3549 { c } { T , F , TF }

```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 is the code to insert, #2 is the *next* case to check on and #3 is all of the rest of the cases code. That means that #4 is the **true** branch code, and #5 tidies up the spare \q_mark and the false branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 is empty, #2 is the first \q_mark and so #4 is the **false** code (the true code is mopped up by #3).

```

3550 \cs_new:Npn \__tl_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
3551 { \exp_end: #1 #4 }

```

(End definition for \tl_case:NnTF and others. This function is documented on page 39.)

6.7 Mapping to token lists

\tl_map_function:nN Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker is read immediately and the loop terminated.

```

\__tl_map_function:Nn
3552 \cs_new:Npn \tl_map_function:nN #1#2
3553 {
3554   \__tl_map_function:Nn #2 #1
3555   \q_recursion_tail
3556   \prg_break_point:Nn \tl_map_break: { }
3557 }
3558 \cs_new:Npn \tl_map_function:NN
3559 { \exp_args:No \tl_map_function:nN }
3560 \cs_new:Npn \__tl_map_function:Nn #1#2
3561 {
3562   \quark_if_recursion_tail_break:nN {#2} \tl_map_break:
3563   #1 {#2} \__tl_map_function:Nn #1
3564 }
3565 \cs_generate_variant:Nn \tl_map_function:NN { c }

```

(End definition for \tl_map_function:nN, \tl_map_function:NN, and __tl_map_function:Nn. These functions are documented on page 39.)

`\tl_map_inline:nn` The inline functions are straight forward by now. We use a little trick with the counter
`\tl_map_inline:Nn` `\g__kernel_prg_map_int` to make them nestable. We can also make use of `__tl_-`
`\tl_map_inline:cn` `map_function:Nn` from before.

```

3566 \cs_new_protected:Npn \tl_map_inline:nn #1#2
3567 {
3568   \int_gincr:N \g__kernel_prg_map_int
3569   \cs_gset_protected:cpn
3570     { __tl_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
3571   \exp_args:Nc \__tl_map_function:Nn
3572     { __tl_map_ \int_use:N \g__kernel_prg_map_int :w }
3573     #1 \q_recursion_tail
3574   \prg_break_point:Nn \tl_map_break:
3575     { \int_gdecr:N \g__kernel_prg_map_int }
3576 }
3577 \cs_new_protected:Npn \tl_map_inline:Nn
3578 { \exp_args:No \tl_map_inline:nn }
3579 \cs_generate_variant:Nn \tl_map_inline:Nn { c }

```

(End definition for `\tl_map_inline:nn` and `\tl_map_inline:Nn`. These functions are documented on page 40.)

`\tl_map_variable:nNn` `\tl_map_variable:nNn` `<token list>` `<temp>` `<action>` assigns `<temp>` to each element and
`\tl_map_variable:NNn` executes `<action>`.

```

3580 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
3581 {
3582   \__tl_map_variable:Nnn #2 {#3} #1
3583   \q_recursion_tail
3584   \prg_break_point:Nn \tl_map_break: { }
3585 }
3586 \cs_new_protected:Npn \tl_map_variable:NNn
3587 { \exp_args:No \tl_map_variable:nNn }
3588 \cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
3589 {
3590   \tl_set:Nn #1 {#3}
3591   \quark_if_recursion_tail_break:NN #1 \tl_map_break:
3592   \use:n {#2}
3593   \__tl_map_variable:Nnn #1 {#2}
3594 }
3595 \cs_generate_variant:Nn \tl_map_variable:NNn { c }

```

(End definition for `\tl_map_variable:nNn`, `\tl_map_variable:NNn`, and `__tl_map_variable:Nnn`. These functions are documented on page 40.)

`\tl_map_break:` The break statements use the general `\prg_map_break:Nn`.
`\tl_map_break:n`

```

3596 \cs_new:Npn \tl_map_break:
3597 { \prg_map_break:Nn \tl_map_break: { } }
3598 \cs_new:Npn \tl_map_break:n
3599 { \prg_map_break:Nn \tl_map_break: }

```

(End definition for `\tl_map_break:` and `\tl_map_break:n`. These functions are documented on page 40.)

6.8 Using token lists

\tl_to_str:n Another name for a primitive: defined in l3basics.

\tl_to_str:V 3600 \cs_generate_variant:Nn \tl_to_str:n { V }

(End definition for \tl_to_str:n. This function is documented on page 41.)

\tl_to_str:N These functions return the replacement text of a token list as a string.

\tl_to_str:c 3601 \cs_new:Npn \tl_to_str:N #1 { __kernel_tl_to_str:w \exp_after:wN {#1} }

3602 \cs_generate_variant:Nn \tl_to_str:N { c }

(End definition for \tl_to_str:N. This function is documented on page 41.)

\tl_use:N Token lists which are simply not defined give a clear TeX error here. No such luck for ones equal to \scan_stop: so instead a test is made and if there is an issue an error is forced.

\tl_use:c

3603 \cs_new:Npn \tl_use:N #1
3604 {
3605 \tl_if_exist:NTF #1 {#1}
3606 {
3607 __kernel_msg_expandable_error:nnn
3608 { kernel } { bad-variable } {#1}
3609 }
3610 }
3611 \cs_generate_variant:Nn \tl_use:N { c }

(End definition for \tl_use:N. This function is documented on page 42.)

6.9 Working with the contents of token lists

\tl_count:n Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. __tl_count:n grabs the element and replaces it by +1. The 0 ensures that it works on an empty list.

\tl_count:V

\tl_count:o

\tl_count:N

\tl_count:c

__tl_count:n

3612 \cs_new:Npn \tl_count:n #1
3613 {
3614 \int_eval:n
3615 { 0 \tl_map_function:nN {#1} __tl_count:n }
3616 }
3617 \cs_new:Npn \tl_count:N #1
3618 {
3619 \int_eval:n
3620 { 0 \tl_map_function:NN #1 __tl_count:n }
3621 }
3622 \cs_new:Npn __tl_count:n #1 { + 1 }
3623 \cs_generate_variant:Nn \tl_count:n { V , o }
3624 \cs_generate_variant:Nn \tl_count:N { c }

(End definition for \tl_count:n, \tl_count:N, and __tl_count:n. These functions are documented on page 42.)

\tl_reverse_items:n Reversal of a token list is done by taking one item at a time and putting it after \q_stop.

__tl_reverse_items:nwNwn

__tl_reverse_items:wn

3625 \cs_new:Npn \tl_reverse_items:n #1
3626 {
3627 __tl_reverse_items:nwNwn #1 ?

```

3628     \q_mark \_tl_reverse_items:nwNwn
3629     \q_mark \_tl_reverse_items:wn
3630     \q_stop { }
3631   }
3632 \cs_new:Npn \_tl_reverse_items:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
3633 {
3634   #3 #2
3635   \q_mark \_tl_reverse_items:nwNwn
3636   \q_mark \_tl_reverse_items:wn
3637   \q_stop { {#1} #5 }
3638 }
3639 \cs_new:Npn \_tl_reverse_items:wn #1 \q_stop #2
3640 { \exp_not:o { \use_none:n #2 } }

```

(End definition for `\tl_reverse_items:n`, `_tl_reverse_items:nwNwn`, and `_tl_reverse_items:wn`. This function is documented on page 42.)

`\tl_trim_spaces:n` Trimming spaces from around the input is deferred to an internal function whose first argument is the token list to trim, augmented by an initial `\q_mark`, and whose second argument is a *<continuation>*, which receives as a braced argument `\use_none:n \q_mark` *<trimmed token list>*. In the case at hand, we take `\exp_not:o` as our continuation, so that space trimming behaves correctly within an x-type expansion.

```

3641 \cs_new:Npn \tl_trim_spaces:n #1
3642 { \_tl_trim_spaces:nn { \q_mark #1 } \exp_not:o }
3643 \cs_generate_variant:Nn \tl_trim_spaces:n { o }
3644 \cs_new:Npn \tl_trim_spaces_apply:nN #1#2
3645 { \_tl_trim_spaces:nn { \q_mark #1 } { \exp_args:No #2 } }
3646 \cs_generate_variant:Nn \tl_trim_spaces_apply:nN { o }
3647 \cs_new_protected:Npn \tl_trim_spaces:N #1
3648 { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
3649 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
3650 { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
3651 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
3652 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }

```

Trimming spaces from around the input is done using delimited arguments and quarks, and to get spaces at odd places in the definitions, we nest those in `_tl_tmp:w`, which then receives a single space as its argument: `#1` is `␣`. Removing leading spaces is done with `_tl_trim_spaces_auxi:w`, which loops until `\q_mark␣` matches the end of the token list: then `##1` is the token list and `##3` is `_tl_trim_spaces_auxii:w`. This hands the relevant tokens to the loop `_tl_trim_spaces_auxiii:w`, responsible for trimming trailing spaces. The end is reached when `␣ \q_nil` matches the one present in the definition of `\tl_trim_spaces:n`. Then `_tl_trim_spaces_auxiv:w` puts the token list into a group, with `\use_none:n` placed there to gobble a lingering `\q_mark`, and feeds this to the *<continuation>*.

```

3653 \cs_set:Npn \_tl_tmp:w #1
3654 {
3655   \cs_new:Npn \_tl_trim_spaces:nn ##1
3656   {
3657     \_tl_trim_spaces_auxi:w
3658     ##1
3659     \q_nil
3660     \q_mark #1 { }

```

```

3661         \q_mark \_tl_trim_spaces_auxii:w
3662         \_tl_trim_spaces_auxiii:w
3663         #1 \q_nil
3664         \_tl_trim_spaces_auxiv:w
3665     \q_stop
3666 }
3667 \cs_new:Npn \_tl_trim_spaces_auxi:w ##1 \q_mark #1 ##2 \q_mark ##3
3668 {
3669     ##3
3670     \_tl_trim_spaces_auxi:w
3671     \q_mark
3672     ##2
3673     \q_mark #1 {##1}
3674 }
3675 \cs_new:Npn \_tl_trim_spaces_auxii:w
3676     \_tl_trim_spaces_auxi:w \q_mark \q_mark ##1
3677 {
3678     \_tl_trim_spaces_auxiii:w
3679     ##1
3680 }
3681 \cs_new:Npn \_tl_trim_spaces_auxiii:w ##1 #1 \q_nil ##2
3682 {
3683     ##2
3684     ##1 \q_nil
3685     \_tl_trim_spaces_auxiii:w
3686 }
3687 \cs_new:Npn \_tl_trim_spaces_auxiv:w ##1 \q_nil ##2 \q_stop ##3
3688     { ##3 { \use_none:n ##1 } }
3689 }
3690 \_tl_tmp:w { ~ }

```

(End definition for `\tl_trim_spaces:n` and others. These functions are documented on page 43.)

`\tl_sort:Nn` Implemented in `l3sort`.

`\tl_sort:cn`

`\tl_gsort:Nn` (End definition for `\tl_sort:Nn`, `\tl_gsort:Nn`, and `\tl_sort:nN`. These functions are documented on page 43.)

`\tl_gsort:cn`

`\tl_sort:nN`

6.10 Token by token changes

`\q__tl_act_mark`

`\q__tl_act_stop`

The `__tl_act_...` functions may be applied to any token list. Hence, we use two private quarks, to allow any token, even quarks, in the token list. Only `\q__tl_act_mark` and `\q__tl_act_stop` may not appear in the token lists manipulated by `__tl_act:NNNnn` functions. No quark module yet, so do things by hand.

```

3691 \cs_new_nopar:Npn \q__tl_act_mark { \q__tl_act_mark }
3692 \cs_new_nopar:Npn \q__tl_act_stop { \q__tl_act_stop }

```

(End definition for `\q__tl_act_mark` and `\q__tl_act_stop`.)

`__tl_act:NNNnn`

`__tl_act_output:n`

`__tl_act_reverse_output:n`

`__tl_act_loop:w`

`__tl_act_normal:NwnNNN`

`__tl_act_group:nwnNNN`

`__tl_act_space:wwnNNN`

`__tl_act_end:w`

To help control the expansion, `__tl_act:NNNnn` should always be proceeded by `\exp:w` and ends by producing `\exp_end:` once the result has been obtained. Then loop over tokens, groups, and spaces in #5. The marker `\q__tl_act_mark` is used both to avoid losing outer braces and to detect the end of the token list more easily. The result is stored as an argument for the dummy function `__tl_act_result:n`.

```

3693 \cs_new:Npn \__tl_act:NNNnn #1#2#3#4#5
3694 {
3695   \group_align_safe_begin:
3696   \__tl_act_loop:w #5 \q__tl_act_mark \q__tl_act_stop
3697   {#4} #1 #2 #3
3698   \__tl_act_result:n { }
3699 }

```

In the loop, we check how the token list begins and act accordingly. In the “normal” case, we may have reached `\q__tl_act_mark`, the end of the list. Then leave `\exp_end:` and the result in the input stream, to terminate the expansion of `\exp:w`. Otherwise, apply the relevant function to the “arguments”, #3 and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with a group or with a space. Some extra work is needed to make `__tl_act_space:wnnNNN` gobble the space.

```

3700 \cs_new:Npn \__tl_act_loop:w #1 \q__tl_act_stop
3701 {
3702   \tl_if_head_is_N_type:nTF {#1}
3703   { \__tl_act_normal:NwnNNN }
3704   {
3705     \tl_if_head_is_group:nTF {#1}
3706     { \__tl_act_group:nwnNNN }
3707     { \__tl_act_space:wnnNNN }
3708   }
3709   #1 \q__tl_act_stop
3710 }
3711 \cs_new:Npn \__tl_act_normal:NwnNNN #1 #2 \q__tl_act_stop #3#4
3712 {
3713   \if_meaning:w \q__tl_act_mark #1
3714   \exp_after:wN \__tl_act_end:wn
3715   \fi:
3716   #4 {#3} #1
3717   \__tl_act_loop:w #2 \q__tl_act_stop
3718   {#3} #4
3719 }
3720 \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
3721 { \group_align_safe_end: \exp_end: #2 }
3722 \cs_new:Npn \__tl_act_group:nwnNNN #1 #2 \q__tl_act_stop #3#4#5
3723 {
3724   #5 {#3} {#1}
3725   \__tl_act_loop:w #2 \q__tl_act_stop
3726   {#3} #4 #5
3727 }
3728 \exp_last_unbraced:NNo
3729 \cs_new:Npn \__tl_act_space:wnnNNN \c_space_tl #1 \q__tl_act_stop #2#3#4#5
3730 {
3731   #5 {#2}
3732   \__tl_act_loop:w #1 \q__tl_act_stop
3733   {#2} #3 #4 #5
3734 }

```

Typically, the output is done to the right of what was already output, using `__tl_act_output:n`, but for the `__tl_act_reverse` functions, it should be done to the left.

```

3735 \cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
3736 { #2 \__tl_act_result:n { #3 #1 } }

```

```

3737 \cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
3738 { #2 \__tl_act_result:n { #1 #3 } }

```

(End definition for __tl_act:NNNnn and others.)

\tl_reverse:n The goal here is to reverse without losing spaces nor braces. This is done using the
\tl_reverse:o general internal function __tl_act:NNNnn. Spaces and “normal” tokens are output on
\tl_reverse:V the left of the current output. Grouped tokens are output to the left but without any
__tl_reverse_normal:nN reversal within the group. All of the internal functions here drop one argument: this is
__tl_reverse_group_preserve:nn needed by __tl_act:NNNnn when changing case (to record which direction the change
__tl_reverse_space:n is in), but not when reversing the tokens.

```

3739 \cs_new:Npn \tl_reverse:n #1
3740 {
3741   \__kernel_exp_not:w \exp_after:wN
3742   {
3743     \exp:w
3744     \__tl_act:NNNnn
3745     \__tl_reverse_normal:nN
3746     \__tl_reverse_group_preserve:nn
3747     \__tl_reverse_space:n
3748     { }
3749     {#1}
3750   }
3751 }
3752 \cs_generate_variant:Nn \tl_reverse:n { o , V }
3753 \cs_new:Npn \__tl_reverse_normal:nN #1#2
3754 { \__tl_act_reverse_output:n {#2} }
3755 \cs_new:Npn \__tl_reverse_group_preserve:nn #1#2
3756 { \__tl_act_reverse_output:n { {#2} } }
3757 \cs_new:Npn \__tl_reverse_space:n #1
3758 { \__tl_act_reverse_output:n { ~ } }

```

(End definition for \tl_reverse:n and others. This function is documented on page 42.)

\tl_reverse:N This reverses the list, leaving \exp_stop_f: in front, which stops the f-expansion.
\tl_reverse:c
\tl_greverse:N
\tl_greverse:c

```

3759 \cs_new_protected:Npn \tl_reverse:N #1
3760 { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
3761 \cs_new_protected:Npn \tl_greverse:N #1
3762 { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
3763 \cs_generate_variant:Nn \tl_reverse:N { c }
3764 \cs_generate_variant:Nn \tl_greverse:N { c }

```

(End definition for \tl_reverse:N and \tl_greverse:N. These functions are documented on page 42.)

6.11 The first token from a token list

\tl_head:N Finding the head of a token list expandably always strips braces, which is fine as this
\tl_head:n is consistent with for example mapping to a list. The empty brace groups in \tl_
\tl_head:V head:n ensure that a blank argument gives an empty result. The result is returned
\tl_head:v within the \unexpanded primitive. The approach here is to use \if_false: to allow
\tl_head:f us to use } as the closing delimiter: this is the only safe choice, as any other token
__tl_head_auxi:nw would not be able to parse it’s own code. Using a marker, we can see if what we are
__tl_head_auxii:n grabbing is exactly the marker, or there is anything else to deal with. Is there is, there
\tl_head:w
\tl_tail:N
\tl_tail:n
\tl_tail:V
\tl_tail:v
\tl_tail:f

is a loop. If not, tidy up and leave the item in the output stream. More detail in <http://tex.stackexchange.com/a/70168>.

```

3765 \cs_new:Npn \tl_head:n #1
3766 {
3767   \__kernel_exp_not:w
3768   \if_false: { \fi: \__tl_head_auxi:nw #1 { } \q_stop }
3769 }
3770 \cs_new:Npn \__tl_head_auxi:nw #1#2 \q_stop
3771 {
3772   \exp_after:wN \__tl_head_auxii:n \exp_after:wN {
3773     \if_false: } \fi: {#1}
3774 }
3775 \cs_new:Npn \__tl_head_auxii:n #1
3776 {
3777   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
3778   \__kernel_tl_to_str:w \exp_after:wN { \use_none:n #1 } \q_nil
3779   \exp_after:wN \use_i:nn
3780   \else:
3781     \exp_after:wN \use_ii:nn
3782   \fi:
3783   {#1}
3784   { \if_false: { \fi: \__tl_head_auxi:nw #1 } }
3785 }
3786 \cs_generate_variant:Nn \tl_head:n { V , v , f }
3787 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
3788 \cs_new:Npn \tl_head:N { \exp_args:No \tl_head:n }

```

To correctly leave the tail of a token list, it's important *not* to absorb any of the tail part as an argument. For example, the simple definition

```

\cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
\cs_new:Npn \tl_tail:w #1#2 \q_stop

```

would give the wrong result for `\tl_tail:n { a { bc } }` (the braces would be stripped). Thus the only safe way to proceed is to first check that there is an item to grab (*i.e.* that the argument is not blank) and assuming there is to dispose of the first item. As with `\tl_head:n`, the result is protected from further expansion by `\unexpanded`. While we could optimise the test here, this would leave some tokens “banned” in the input, which we do not have with this definition.

```

3789 \cs_new:Npn \tl_tail:n #1
3790 {
3791   \__kernel_exp_not:w
3792   \tl_if_blank:nTF {#1}
3793     { { } }
3794     { \exp_after:wN { \use_none:n #1 } }
3795 }
3796 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
3797 \cs_new:Npn \tl_tail:N { \exp_args:No \tl_tail:n }

```

(End definition for `\tl_head:N` and others. These functions are documented on page 44.)

Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

```

\tl_if_head_eq_meaning_p:nN
\tl_if_head_eq_meaning:nNTF
\tl_if_head_eq_charcode_p:nN
\tl_if_head_eq_charcode:nNTF
\tl_if_head_eq_charcode_p:fN
\tl_if_head_eq_charcode:fNTF
\tl_if_head_eq_catcode_p:nN
\tl_if_head_eq_catcode:nNTF

```

Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, `\tl_head:w` yields the first token of the token list, then passed to `\exp_not:N`.

```
\if_charcode:w
  \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
  \exp_not:N #2
```

The two first special cases are detected by testing if the token list starts with an N-type token (the extra ? sends empty token lists to the `true` branch of this test). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character). An empty argument results in `\tl_head:w` leaving two tokens: ? which is taken in the `\if_charcode:w` test, and `\use_none:nn`, which ensures that `\prg_return_false:` is returned regardless of whether the charcode test was true or false.

```
3798 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
3799 {
3800   \if_charcode:w
3801     \exp_not:N #2
3802     \tl_if_head_is_N_type:nTF { #1 ? }
3803     {
3804       \exp_after:wN \exp_not:N
3805       \tl_head:w #1 { ? \use_none:nn } \q_stop
3806     }
3807     { \str_head:n {#1} }
3808     \prg_return_true:
3809   \else:
3810     \prg_return_false:
3811   \fi:
3812 }
3813 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_charcode:nN
3814 { f } { p , TF , T , F }
```

For `\tl_if_head_eq_catcode:nN`, again we detect special cases with a `\tl_if_head_is_N_type:n`. Then we need to test if the first token is a begin-group token or an explicit space token, and produce the relevant token, either `\c_group_begin_token` or `\c_space_token`. Again, for an empty argument, a hack is used, removing `\prg_return_true:` and `\else:` with `\use_none:nn` in case the catcode test with the (arbitrarily chosen) ? is true.

```
3815 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
3816 {
3817   \if_catcode:w
3818     \exp_not:N #2
3819     \tl_if_head_is_N_type:nTF { #1 ? }
3820     {
3821       \exp_after:wN \exp_not:N
3822       \tl_head:w #1 { ? \use_none:nn } \q_stop
3823     }
3824     {
3825       \tl_if_head_is_group:nTF {#1}
3826       { \c_group_begin_token }
3827       { \c_space_token }
3828     }
3829   }
3830 }
```



```

3829     \prg_return_true:
3830 \else:
3831     \prg_return_false:
3832 \fi:
3833 }

```

For `\tl_if_head_eq_meaning:nN`, again, detect special cases. In the normal case, use `\tl_head:w`, with no `\exp_not:N` this time, since `\if_meaning:w` causes no expansion. With an empty argument, the test is true, and `\use_none:nnn` removes #2 and the usual `\prg_return_true:` and `\else:`. In the special cases, we know that the first token is a character, hence `\if_charcode:w` and `\if_catcode:w` together are enough. We combine them in some order, hopefully faster than the reverse. Tests are not nested because the arguments may contain unmatched primitive conditionals.

```

3834 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
3835 {
3836   \tl_if_head_is_N_type:nTF { #1 ? }
3837     { \__tl_if_head_eq_meaning_normal:nN }
3838     { \__tl_if_head_eq_meaning_special:nN }
3839   {#1} #2
3840 }
3841 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
3842 {
3843   \exp_after:wN \if_meaning:w
3844   \tl_head:w #1 { ?? \use_none:nnn } \q_stop #2
3845   \prg_return_true:
3846 \else:
3847   \prg_return_false:
3848 \fi:
3849 }
3850 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
3851 {
3852   \if_charcode:w \str_head:n {#1} \exp_not:N #2
3853   \exp_after:wN \use:n
3854 \else:
3855   \prg_return_false:
3856   \exp_after:wN \use_none:n
3857 \fi:
3858 {
3859   \if_catcode:w \exp_not:N #2
3860     \tl_if_head_is_group:nTF {#1}
3861       { \c_group_begin_token }
3862       { \c_space_token }
3863     \prg_return_true:
3864 \else:
3865   \prg_return_false:
3866 \fi:
3867 }
3868 }

```

(End definition for `\tl_if_head_eq_meaning:nNTF` and others. These functions are documented on page 45.)

`\tl_if_head_is_N_type_p:n` A token list can be empty, can start with an explicit space character (catcode 10 and
`\tl_if_head_is_N_type:nTF` charcode 32), can start with a begin-group token (catcode 1), or start with an N-type
`__tl_if_head_is_N_type:w`

argument. In the first two cases, the line involving `__tl_if_head_is_N_type:w` produces `^` (and otherwise nothing). In the third case (begin-group token), the lines involving `\exp_after:wN` produce a single closing brace. The category code test is thus true exactly in the fourth case, which is what we want. One cannot optimize by moving one of the `*` to the beginning: if `#1` contains primitive conditionals, all of its occurrences must be dealt with before the `\if_catcode:w` tries to skip the `true` branch of the conditional.

```

3869 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
3870 {
3871   \if_catcode:w
3872     \if_false: { \fi: \__tl_if_head_is_N_type:w ? #1 ~ }
3873     \exp_after:wN \use_none:n
3874     \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
3875     * *
3876     \prg_return_true:
3877   \else:
3878     \prg_return_false:
3879   \fi:
3880 }
3881 \cs_new:Npn \__tl_if_head_is_N_type:w #1 ~
3882 {
3883   \tl_if_empty:oTF { \use_none:n #1 } { ^ } { }
3884   \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
3885 }

```

(End definition for `\tl_if_head_is_N_type:nTF` and `__tl_if_head_is_N_type:w`. This function is documented on page 45.)

`\tl_if_head_is_group:p:n` Pass the first token of `#1` through `\token_to_str:N`, then check for the brace balance.
`\tl_if_head_is_group:nTF` The extra `?` caters for an empty argument.⁶

```

3886 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
3887 {
3888   \if_catcode:w
3889     \exp_after:wN \use_none:n
3890     \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
3891     * *
3892     \prg_return_false:
3893   \else:
3894     \prg_return_true:
3895   \fi:
3896 }

```

(End definition for `\tl_if_head_is_group:nTF`. This function is documented on page 45.)

`\tl_if_head_is_space:p:n` The auxiliary's argument is all that is before the first explicit space in `?#1?~`. If that
`\tl_if_head_is_space:nTF` is a single `?` the test yields `true`. Otherwise, that is more than one token, and the
`__tl_if_head_is_space:w` test yields `false`. The work is done within braces (with an `\if_false: { \fi: ... }` construction) both to hide potential alignment tab characters from \TeX in a table, and to allow for removing what remains of the token list after its first space. The `\exp:w` and `\exp_end:` ensure that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

⁶Bruno: this could be made faster, but we don't: if we hope to ever have an e-type argument, we need all brace "tricks" to happen in one step of expansion, keeping the token list brace balanced at all times.

```

3897 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
3898 {
3899   \exp:w \if_false: { \fi:
3900     \__tl_if_head_is_space:w ? #1 ? ~ }
3901 }
3902 \cs_new:Npn \__tl_if_head_is_space:w #1 ~
3903 {
3904   \tl_if_empty:oTF { \use_none:n #1 }
3905   { \exp_after:wN \exp_end: \exp_after:wN \prg_return_true: }
3906   { \exp_after:wN \exp_end: \exp_after:wN \prg_return_false: }
3907   \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
3908 }

```

(End definition for `\tl_if_head_is_space:nTF` and `__tl_if_head_is_space:w`. This function is documented on page 45.)

6.12 Using a single item

```

\tl_item:nn The idea here is to find the offset of the item from the left, then use a loop to grab
\tl_item:Nn the correct item. If the resulting offset is too large, then \quark_if_recursion_tail_
\tl_item:cn stop:n terminates the loop, and returns nothing at all.
__tl_item_aux:nn 3909 \cs_new:Npn \tl_item:nn #1#2
__tl_item:nn 3910 {
3911   \exp_args:Nf \__tl_item:nn
3912   { \exp_args:Nf \__tl_item_aux:nn { \int_eval:n {#2} } {#1} }
3913   #1
3914   \q_recursion_tail
3915   \prg_break_point:
3916 }
3917 \cs_new:Npn \__tl_item_aux:nn #1#2
3918 {
3919   \int_compare:nNnTF {#1} < 0
3920   { \int_eval:n { \tl_count:n {#2} + 1 + #1 } }
3921   {#1}
3922 }
3923 \cs_new:Npn \__tl_item:nn #1#2
3924 {
3925   \quark_if_recursion_tail_break:nN {#2} \prg_break:
3926   \int_compare:nNnTF {#1} = 1
3927   { \prg_break:n { \exp_not:n {#2} } }
3928   { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
3929 }
3930 \cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
3931 \cs_generate_variant:Nn \tl_item:Nn { c }

```

(End definition for `\tl_item:nn` and others. These functions are documented on page 46.)

6.13 Viewing token lists

```

\tl_show:N Showing token list variables is done after checking that the variable is defined (see \__-
\tl_show:c kernel_register_show:N).
\tl_log:N 3932 \cs_new_protected:Npn \tl_show:N { \__tl_show:NN \tl_show:n }
\tl_log:c 3933 \cs_generate_variant:Nn \tl_show:N { c }
__tl_show:NN 3934 \cs_new_protected:Npn \tl_log:N { \__tl_show:NN \tl_log:n }

```

```

3935 \cs_generate_variant:Nn \tl_log:N { c }
3936 \cs_new_protected:Npn \__tl_show:NN #1#2
3937 {
3938   \__kernel_chk_defined:NT #2
3939   { \exp_args:Nx #1 { \token_to_str:N #2 = \exp_not:o {#2} } } }
3940 }

```

(End definition for `\tl_show:N`, `\tl_log:N`, and `__tl_show:NN`. These functions are documented on page 46.)

`\tl_show:n` Many show functions are based on `\tl_show:n`. The argument of `\tl_show:n` is line-wrapped using `\iow_wrap:nnnN` but with a leading `>~` and trailing period, both removed before passing the wrapped text to the `\showtokens` primitive. This primitive shows the result with a leading `>~` and trailing period.

The token list `\l__tl_internal_a_tl` containing the result of all these manipulations is displayed to the terminal using `\etex_showtokens:D` and an odd `\exp_after:wN` which expand the closing brace to improve the output slightly. The calls to `__kernel_iow_with:Nnn` ensure that the `\newlinechar` is set to 10 so that the `\iow_newline:` inserted by the line-wrapping code are correctly recognized by T_EX, and that `\errorcontextlines` is -1 to avoid printing irrelevant context.

```

3941 \cs_new_protected:Npn \tl_show:n #1
3942 { \iow_wrap:nnnN { >~ \tl_to_str:n {#1} . } { } { } \__tl_show:n }
3943 \cs_new_protected:Npn \__tl_show:n #1
3944 {
3945   \tl_set:Nf \l__tl_internal_a_tl { \__tl_show:w #1 \q_stop }
3946   \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
3947   {
3948     \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
3949     {
3950       \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
3951       { \exp_after:wN \l__tl_internal_a_tl }
3952     }
3953   }
3954 }
3955 \cs_new:Npn \__tl_show:w #1 > #2 . \q_stop {#2}

```

(End definition for `\tl_show:n`, `__tl_show:n`, and `__tl_show:w`. This function is documented on page 46.)

`\tl_log:n` Logging is much easier, simply line-wrap. The `>~` and trailing period is there to match the output of `\tl_show:n`.

```

3956 \cs_new_protected:Npn \tl_log:n #1
3957 { \iow_wrap:nnnN { > ~ \tl_to_str:n {#1} . } { } { } \iow_log:n }

```

(End definition for `\tl_log:n`. This function is documented on page 46.)

6.14 Scratch token lists

`\g_tmpa_tl` Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```

3958 \tl_new:N \g_tmpa_tl
3959 \tl_new:N \g_tmpb_tl

```

(End definition for `\g_tmpa_tl` and `\g_tmpb_tl`. These variables are documented on page 47.)

`\l_tmpa_tl` These are local temporary token list variables. Be sure not to assume that the value you
`\l_tmpb_tl` put into them will survive for long—see discussion above.

```
3960 \tl_new:N \l_tmpa_tl
3961 \tl_new:N \l_tmpb_tl
```

(End definition for `\l_tmpa_tl` and `\l_tmpb_tl`. These variables are documented on page 47.)

```
3962 </initex | package>
```

7 l3str implementation

```
3963 <(*initex | package>
```

```
3964 <@@=str>
```

7.1 Creating and setting string variables

`\str_new:N` A string is simply a token list. The full mapping system isn't set up yet so do things by
`\str_new:c` hand.

```
\str_use:N 3965 \group_begin:
\str_use:c 3966 \cs_set_protected:Npn \__str_tmp:n #1
\str_clear:N 3967 {
\str_clear:c 3968 \tl_if_blank:nF {#1}
\str_gclear:N 3969 {
\str_gclear:c 3970 \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
\str_clear_new:N 3971 \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
\str_clear_new:c 3972 \__str_tmp:n
\str_gclear_new:N 3973 }
\str_gclear_new:c 3974 }
\str_set_eq:NN 3975 \__str_tmp:n
\str_set_eq:cN 3976 { new }
\str_set_eq:Nc 3977 { use }
\str_set_eq:Nc 3978 { clear }
\str_set_eq:cc 3979 { gclear }
\str_gset_eq:NN 3980 { clear_new }
\str_gset_eq:cN 3981 { gclear_new }
\str_gset_eq:Nc 3982 { }
\str_gset_eq:cc 3983 \group_end:
\str_concat:NNN 3984 \cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN
\str_concat:ccc 3985 \cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN
\str_gconcat:NNN 3986 \cs_generate_variant:Nn \str_set_eq:NN { c , Nc , cc }
\str_gconcat:ccc 3987 \cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }
3988 \cs_new_eq:NN \str_concat:NNN \tl_concat:NNN
3989 \cs_new_eq:NN \str_gconcat:NNN \tl_gconcat:NNN
3990 \cs_generate_variant:Nn \str_concat:NNN { ccc }
3991 \cs_generate_variant:Nn \str_gconcat:NNN { ccc }
```

(End definition for `\str_new:N` and others. These functions are documented on page 48.)

`\str_set:Nn` Simply convert the token list inputs to `<strings>`.

```
\str_set:Nx 3992 \group_begin:
\str_set:cn 3993 \cs_set_protected:Npn \__str_tmp:n #1
\str_set:cx 3994 {
```

`\str_gset:Nn`

`\str_gset:Nx`

`\str_gset:cn`

`\str_gset:cx`

`\str_const:Nn`

`\str_const:Nx`

`\str_const:cn`

`\str_const:cx`

`\str_put_left:Nn`

`\str_put_left:Nx`

```

3995     \tl_if_blank:nF {#1}
3996     {
3997         \cs_new_protected:cpx { str_ #1 :Nn } ##1##2
3998         {
3999             \exp_not:c { tl_ #1 :Nx } ##1
4000             { \exp_not:N \tl_to_str:n {##2} }
4001         }
4002         \cs_generate_variant:cn { str_ #1 :Nn } { Nx , cn , cx }
4003         \__str_tmp:n
4004     }
4005 }
4006 \__str_tmp:n
4007 { set }
4008 { gset }
4009 { const }
4010 { put_left }
4011 { gput_left }
4012 { put_right }
4013 { gput_right }
4014 { }
4015 \group_end:

```

(End definition for `\str_set:Nn` and others. These functions are documented on page 49.)

7.2 Modifying string variables

`\str_replace_all:Nnn` Start by applying `\tl_to_str:n` to convert the old and new token lists to strings, and
`\str_replace_all:cnn` also apply `\tl_to_str:N` to avoid any issues if we are fed a token list variable. Then the
`\str_greplace_all:Nnn` code is a much simplified version of the token list code because neither the delimiter nor
`\str_greplace_all:cnn` the replacement can contain macro parameters or braces. The delimiter `\q_mark` cannot
`\str_replace_once:Nnn` appear in the string to edit so it is used in all cases. Some x-expansion is unnecessary.
`\str_replace_once:cnn` There is no need to avoid losing braces nor to protect against expansion. The ending
`\str_greplace_once:Nnn` code is much simplified and does not need to hide in braces.
`\str_greplace_once:cnn`

```

4016 \cs_new_protected:Npn \str_replace_once:Nnn
4017 { \__str_replace:NNNnn \prg_do_nothing: \tl_set:Nx }
4018 \cs_new_protected:Npn \str_greplace_once:Nnn
4019 { \__str_replace:NNNnn \prg_do_nothing: \tl_gset:Nx }
4020 \cs_new_protected:Npn \str_replace_all:Nnn
4021 { \__str_replace:NNNnn \__str_replace_next:w \tl_set:Nx }
4022 \cs_new_protected:Npn \str_greplace_all:Nnn
4023 { \__str_replace:NNNnn \__str_replace_next:w \tl_gset:Nx }
4024 \cs_generate_variant:Nn \str_replace_once:Nnn { c }
4025 \cs_generate_variant:Nn \str_greplace_once:Nnn { c }
4026 \cs_generate_variant:Nn \str_replace_all:Nnn { c }
4027 \cs_generate_variant:Nn \str_greplace_all:Nnn { c }
4028 \cs_new_protected:Npn \__str_replace:NNNnn #1#2#3#4#5
4029 {
4030     \tl_if_empty:nTF {#4}
4031     {
4032         \__kernel_msg_error:nxx { kernel } { empty-search-pattern } {#5}
4033     }
4034     {
4035         \use:x

```

```

4036     {
4037         \exp_not:n { \__str_replace_aux:NNNnnn #1 #2 #3 }
4038         { \tl_to_str:N #3 }
4039         { \tl_to_str:n {#4} } { \tl_to_str:n {#5} }
4040     }
4041 }
4042 }
4043 \cs_new_protected:Npn \__str_replace_aux:NNNnnn #1#2#3#4#5#6
4044 {
4045     \cs_set:Npn \__str_replace_next:w ##1 #5 { ##1 #6 #1 }
4046     #2 #3
4047     {
4048         \__str_replace_next:w
4049         #4
4050         \use_none_delimit_by_q_stop:w
4051         #5
4052         \q_stop
4053     }
4054 }
4055 \cs_new_eq:NN \__str_replace_next:w ?

```

(End definition for `\str_replace_all:Nnn` and others. These functions are documented on page 50.)

```

\str_remove_once:Nn Removal is just a special case of replacement.
\str_remove_once:cn 4056 \cs_new_protected:Npn \str_remove_once:Nn #1#2
\str_gremove_once:Nn 4057 { \str_replace_once:Nnn #1 {#2} { } }
\str_gremove_once:cn 4058 \cs_new_protected:Npn \str_gremove_once:Nn #1#2
4059 { \str_greplace_once:Nnn #1 {#2} { } }
4060 \cs_generate_variant:Nn \str_remove_once:Nn { c }
4061 \cs_generate_variant:Nn \str_gremove_once:Nn { c }

```

(End definition for `\str_remove_once:Nn` and `\str_gremove_once:Nn`. These functions are documented on page 50.)

```

\str_remove_all:Nn Removal is just a special case of replacement.
\str_remove_all:cn 4062 \cs_new_protected:Npn \str_remove_all:Nn #1#2
\str_gremove_all:Nn 4063 { \str_replace_all:Nnn #1 {#2} { } }
\str_gremove_all:cn 4064 \cs_new_protected:Npn \str_gremove_all:Nn #1#2
4065 { \str_greplace_all:Nnn #1 {#2} { } }
4066 \cs_generate_variant:Nn \str_remove_all:Nn { c }
4067 \cs_generate_variant:Nn \str_gremove_all:Nn { c }

```

(End definition for `\str_remove_all:Nn` and `\str_gremove_all:Nn`. These functions are documented on page 50.)

7.3 String comparisons

```

\str_if_empty_p:N More copy-paste!
\str_if_empty_p:c 4068 \prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N
\str_if_empty:NTF 4069 { p , T , F , TF }
\str_if_empty:cTF 4070 \prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c
\str_if_exist_p:N 4071 { p , T , F , TF }
\str_if_exist_p:c 4072 \prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N
\str_if_exist:NTF 4073 { p , T , F , TF }
\str_if_exist:cTF 4074 \prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c
4075 { p , T , F , TF }

```

(End definition for `\str_if_empty:NTF` and `\str_if_exist:NTF`. These functions are documented on page 50.)

`__str_if_eq_x:nn` `__str_escape_x:n` String comparisons rely on the primitive `\(pdf)strcmp` if available: LuaTeX does not have it, so emulation is required. As the net result is that we do not *always* use the primitive, the correct approach is to wrap up in a function with defined behaviour. That's done by providing a wrapper and then redefining in the LuaTeX case. Note that the necessary Lua code is loaded in `l3bootstrap`. The need to detokenize and force expansion of input arises from the case where a `#` token is used in the input, e.g. `__str_if_eq_x:nn {#} { \tl_to_str:n {#} }`, which otherwise would fail as `\luatex_luaescapestring:D` does not double such tokens.

```

4076 \cs_new:Npn \__str_if_eq_x:nn #1#2 { \pdfstrcmp:D {#1} {#2} }
4077 \cs_if_exist:NT \luatex luatexversion:D
4078 {
4079   \cs_set_eq:NN \lua_escape_x:n \luatex_luaescapestring:D
4080   \cs_set_eq:NN \lua_now_x:n \luatex_directlua:D
4081   \cs_set:Npn \__str_if_eq_x:nn #1#2
4082   {
4083     \lua_now_x:n
4084     {
4085       l3kernel_strcmp
4086       (
4087         " \__str_escape_x:n {#1} " ,
4088         " \__str_escape_x:n {#2} "
4089       )
4090     }
4091   }
4092   \cs_new:Npn \__str_escape_x:n #1
4093   {
4094     \lua_escape_x:n
4095     { \__kernel_tl_to_str:w \use_x:n { {#1} } }
4096   }
4097 }

```

(End definition for `__str_if_eq_x:nn` and `__str_escape_x:n`.)

`\str_if_eq_p:nn` `\str_if_eq_p:Vn` `\str_if_eq_p:on` `\str_if_eq_p:nV` `\str_if_eq_p:no` `\str_if_eq_p:VV` `\str_if_eq:nnTF` `\str_if_eq:VnTF` `\str_if_eq:onTF` `\str_if_eq:nVTF` `\str_if_eq:noTF` `\str_if_eq:VVTF` `\str_if_eq_x:nn` `\str_if_eq_x:nnTF` Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore make life a bit clearer. The `nn` and `xx` versions are created directly as this is most efficient.

```

4098 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
4099 {
4100   \if_int_compare:w
4101     \__str_if_eq_x:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
4102     = 0 \exp_stop_f:
4103     \prg_return_true: \else: \prg_return_false: \fi:
4104 }
4105 \prg_generate_conditional_variant:Nnn \str_if_eq:nn
4106 { V , o , nV , no , VV } { p , T , F , TF }
4107 \prg_new_conditional:Npnn \str_if_eq_x:nn #1#2 { p , T , F , TF }
4108 {
4109   \if_int_compare:w \__str_if_eq_x:nn {#1} {#2} = 0 \exp_stop_f:
4110   \prg_return_true: \else: \prg_return_false: \fi:
4111 }

```


(End definition for `\str_if_eq:nnTF` and `\str_if_eq_x:nnTF`. These functions are documented on page 51.)

`\str_if_eq_p:NN` Note that `\str_if_eq:NN` is different from `\tl_if_eq:NN` because it needs to ignore category codes.

```

\str_if_eq_p:Nc
\str_if_eq_p:cN
\str_if_eq_p:cc
\str_if_eq:NNTF
\str_if_eq:NcTF
\str_if_eq:cNTF
\str_if_eq:ccTF
4112 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
4113 {
4114   \if_int_compare:w
4115     \__str_if_eq_x:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
4116     = 0 \exp_stop_f: \prg_return_true: \else: \prg_return_false: \fi:
4117 }
4118 \prg_generate_conditional_variant:Nnn \str_if_eq:NN
4119 { c , Nc , cc } { T , F , TF , p }
```

(End definition for `\str_if_eq:NNTF`. This function is documented on page 51.)

`\str_if_in:NnTF` Everything here needs to be detokenized but beyond that it is a simple token list test.
`\str_if_in:cnTF` It would be faster to fine-tune the T, F, TF variants by calling the appropriate variant of
`\str_if_in:nnTF` `\tl_if_in:nnTF` directly but that takes more code.

```

4120 \prg_new_protected_conditional:Npnn \str_if_in:Nn #1#2 { T , F , TF }
4121 {
4122   \use:x
4123   { \tl_if_in:nnTF { \tl_to_str:N #1 } { \tl_to_str:n {#2} } }
4124   { \prg_return_true: } { \prg_return_false: }
4125 }
4126 \prg_generate_conditional_variant:Nnn \str_if_in:Nn
4127 { c } { T , F , TF }
4128 \prg_new_protected_conditional:Npnn \str_if_in:nn #1#2 { T , F , TF }
4129 {
4130   \use:x
4131   { \tl_if_in:nnTF { \tl_to_str:n {#1} } { \tl_to_str:n {#2} } }
4132   { \prg_return_true: } { \prg_return_false: }
4133 }
```

(End definition for `\str_if_in:NnTF` and `\str_if_in:nnTF`. These functions are documented on page 51.)

`\str_case:nn` Much the same as `\tl_case:nn(TF)` here: just a change in the internal comparison.

```

\str_case:on
\str_case:nV
\str_case:nv
\str_case:nnTF
\str_case:onTF
\str_case:nVTF
\str_case:nvTF
\str_case_x:nn
\str_case_x:nnTF
\__str_case:nnTF
\__str_case_x:nnTF
\__str_case:nw
\__str_case_x:nw
\__str_case_end:nw
4134 \cs_new:Npn \str_case:nn #1#2
4135 {
4136   \exp:w
4137   \__str_case:nnTF {#1} {#2} { } { }
4138 }
4139 \cs_new:Npn \str_case:nnT #1#2#3
4140 {
4141   \exp:w
4142   \__str_case:nnTF {#1} {#2} {#3} { }
4143 }
4144 \cs_new:Npn \str_case:nnF #1#2
4145 {
4146   \exp:w
4147   \__str_case:nnTF {#1} {#2} { }
4148 }
4149 \cs_new:Npn \str_case:nnTF #1#2
```

```

4150 {
4151   \exp:w
4152   \__str_case:nnTF {#1} {#2}
4153 }
4154 \cs_new:Npn \__str_case:nnTF #1#2#3#4
4155 { \__str_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
4156 \cs_generate_variant:Nn \str_case:nn { o , nV , nv }
4157 \prg_generate_conditional_variant:Nnn \str_case:nn
4158 { o , nV , nv } { T , F , TF }
4159 \cs_new:Npn \__str_case:nw #1#2#3
4160 {
4161   \str_if_eq:nnTF {#1} {#2}
4162   { \__str_case_end:nw {#3} }
4163   { \__str_case:nw {#1} }
4164 }
4165 \cs_new:Npn \str_case_x:nn #1#2
4166 {
4167   \exp:w
4168   \__str_case_x:nnTF {#1} {#2} { } { }
4169 }
4170 \cs_new:Npn \str_case_x:nnT #1#2#3
4171 {
4172   \exp:w
4173   \__str_case_x:nnTF {#1} {#2} {#3} { }
4174 }
4175 \cs_new:Npn \str_case_x:nnF #1#2
4176 {
4177   \exp:w
4178   \__str_case_x:nnTF {#1} {#2} { }
4179 }
4180 \cs_new:Npn \str_case_x:nnTF #1#2
4181 {
4182   \exp:w
4183   \__str_case_x:nnTF {#1} {#2}
4184 }
4185 \cs_new:Npn \__str_case_x:nnTF #1#2#3#4
4186 { \__str_case_x:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
4187 \cs_new:Npn \__str_case_x:nw #1#2#3
4188 {
4189   \str_if_eq_x:nnTF {#1} {#2}
4190   { \__str_case_end:nw {#3} }
4191   { \__str_case_x:nw {#1} }
4192 }
4193 \cs_new:Npn \__str_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
4194 { \exp_end: #1 #4 }

```

(End definition for `\str_case:nnTF` and others. These functions are documented on page 52.)

7.4 Mapping to strings

`\str_map_function:NN` The inline and variable mappings are similar to the usual token list mappings but start out by turning the argument to an “other string”. Doing the same for the expandable function mapping would require `__kernel_str_to_other:n`, quadratic in the string length. To deal with spaces in that case, `__str_map_function:w` replaces the following

```

\str_map_function:cn
\str_map_function:nN
\str_map_inline:Nn
\str_map_inline:cn
\str_map_inline:nn
\str_map_variable:NNn
\str_map_variable:cNn
\str_map_variable:nNn
\str_map_break:
\str_map_break:n
\__str_map_function:w
\__str_map_function:Nn
\__str_map_inline:NN

```

space by a braced space and a further call to itself. These are received by `__str_map_function:Nn`, which passes the space to `#1` and calls `__str_map_function:w` to deal with the next space. The space before the braced space allows to optimize the `\q_recursion_tail` test. Of course we need to include a trailing space (the question mark is needed to avoid losing the space when `TEX` tokenizes the line). At the cost of about three more auxiliaries this code could get a 9 times speed up by testing only every 9-th character for whether it is `\q_recursion_tail` (also by converting 9 spaces at a time in the `\str_map_function:nN` case).

```

4195 \cs_new:Npn \str_map_function:nN #1#2
4196 {
4197   \exp_after:wN \__str_map_function:w
4198   \exp_after:wN \__str_map_function:Nn \exp_after:wN #2
4199   \__kernel_tl_to_str:w {#1}
4200   \q_recursion_tail ? ~
4201   \prg_break_point:Nn \str_map_break: { }
4202 }
4203 \cs_new:Npn \str_map_function:NN
4204 { \exp_args:No \str_map_function:nN }
4205 \cs_new:Npn \__str_map_function:w #1 ~
4206 { #1 { ~ { ~ } } \__str_map_function:w } }
4207 \cs_new:Npn \__str_map_function:Nn #1#2
4208 {
4209   \if_meaning:w \q_recursion_tail #2
4210   \exp_after:wN \str_map_break:
4211   \fi:
4212   #1 #2 \__str_map_function:Nn #1
4213 }
4214 \cs_generate_variant:Nn \str_map_function:NN { c }
4215 \cs_new_protected:Npn \str_map_inline:nn #1#2
4216 {
4217   \int_gincr:N \g__kernel_prg_map_int
4218   \cs_gset_protected:cpn
4219   { __str_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
4220   \use:x
4221   {
4222     \exp_not:N \__str_map_inline:NN
4223     \exp_not:c { __str_map_ \int_use:N \g__kernel_prg_map_int :w }
4224     \__kernel_str_to_other_fast:n {#1}
4225   }
4226   \q_recursion_tail
4227   \prg_break_point:Nn \str_map_break:
4228   { \int_gdecr:N \g__kernel_prg_map_int }
4229 }
4230 \cs_new_protected:Npn \str_map_inline:Nn
4231 { \exp_args:No \str_map_inline:nn }
4232 \cs_generate_variant:Nn \str_map_inline:Nn { c }
4233 \cs_new:Npn \__str_map_inline:NN #1#2
4234 {
4235   \quark_if_recursion_tail_break:NN #2 \str_map_break:
4236   \exp_args:No #1 { \token_to_str:N #2 }
4237   \__str_map_inline:NN #1
4238 }
4239 \cs_new_protected:Npn \str_map_variable:nNn #1#2#3

```

```

4240 {
4241   \use:x
4242   {
4243     \exp_not:n { \__str_map_variable:NnN #2 {#3} }
4244     \__kernel_str_to_other_fast:n {#1}
4245   }
4246   \q_recursion_tail
4247   \prg_break_point:Nn \str_map_break: { }
4248 }
4249 \cs_new_protected:Npn \str_map_variable:NNn
4250 { \exp_args:No \str_map_variable:nNn }
4251 \cs_new_protected:Npn \__str_map_variable:NnN #1#2#3
4252 {
4253   \quark_if_recursion_tail_break:NN #3 \str_map_break:
4254   \str_set:Nn #1 {#3}
4255   \use:n {#2}
4256   \__str_map_variable:NnN #1 {#2}
4257 }
4258 \cs_generate_variant:Nn \str_map_variable:NNn { c }
4259 \cs_new:Npn \str_map_break:
4260 { \prg_map_break:Nn \str_map_break: { } }
4261 \cs_new:Npn \str_map_break:n
4262 { \prg_map_break:Nn \str_map_break: }

```

(End definition for `\str_map_function:NN` and others. These functions are documented on page 52.)

7.5 Accessing specific characters in a string

```

\__kernel_str_to_other:n
\__str_to_other_loop:w
\__str_to_other_end:w

```

First apply `\tl_to_str:n`, then replace all spaces by “other” spaces, 8 at a time, storing the converted part of the string between the `\q_mark` and `\q_stop` markers. The end is detected when `__str_to_other_loop:w` finds one of the trailing A, distinguished from any contents of the initial token list by their category. Then `__str_to_other_end:w` is called, and finds the result between `\q_mark` and the first A (well, there is also the need to remove a space).

```

4263 \cs_new:Npn \__kernel_str_to_other:n #1
4264 {
4265   \exp_after:wN \__str_to_other_loop:w
4266   \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
4267 }
4268 \group_begin:
4269 \tex_lccode:D ‘\* = ‘\ %
4270 \tex_lccode:D ‘\A = ‘\A %
4271 \tex_lowercase:D
4272 {
4273   \group_end:
4274   \cs_new:Npn \__str_to_other_loop:w
4275     #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \q_stop
4276   {
4277     \if_meaning:w A #8
4278     \__str_to_other_end:w
4279     \fi:
4280     \__str_to_other_loop:w
4281     #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \q_stop
4282   }

```

```

4283 \cs_new:Npn \__str_to_other_end:w \fi: #1 \q_mark #2 * A #3 \q_stop
4284 { \fi: #2 }
4285 }

```

(End definition for `__kernel_str_to_other:n`, `__str_to_other_loop:w`, and `__str_to_other_end:w`.)

```

\__kernel_str_to_other_fast:n
\__kernel_str_to_other_fast_loop:w
\__str_to_other_fast_end:w

```

The difference with `__kernel_str_to_other:n` is that the converted part is left in the input stream, making these commands only restricted-expandable.

```

4286 \cs_new:Npn \__kernel_str_to_other_fast:n #1
4287 {
4288   \exp_after:wN \__str_to_other_fast_loop:w \tl_to_str:n {#1} ~
4289   A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_stop
4290 }
4291 \group_begin:
4292 \tex_lccode:D '\* = '\ %
4293 \tex_lccode:D '\A = '\A %
4294 \tex_lowercase:D
4295 {
4296   \group_end:
4297   \cs_new:Npn \__str_to_other_fast_loop:w
4298     #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
4299     {
4300       \if_meaning:w A #9
4301         \__str_to_other_fast_end:w
4302       \fi:
4303       #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
4304       \__str_to_other_fast_loop:w *
4305     }
4306   \cs_new:Npn \__str_to_other_fast_end:w #1 * A #2 \q_stop {#1}
4307 }

```

(End definition for `__kernel_str_to_other_fast:n`, `__kernel_str_to_other_fast_loop:w`, and `__str_to_other_fast_end:w`.)

```

\str_item:Nn
\str_item:cn
\str_item:nn
\str_item_ignore_spaces:nn
\__str_item:nn
\__str_item:w

```

The `\str_item:nn` hands its argument with spaces escaped to `__str_item:nn`, and makes sure to turn the result back into a proper string (with category code 10 spaces) eventually. The `\str_item_ignore_spaces:nn` function does not escape spaces, which are thus ignored by `__str_item:nn` since everything else is done with undelimited arguments. Evaluate the $\langle index \rangle$ argument #2 and count characters in the string, passing those two numbers to `__str_item:w` for further analysis. If the $\langle index \rangle$ is negative, shift it by the $\langle count \rangle$ to know the how many character to discard, and if that is still negative give an empty result. If the $\langle index \rangle$ is larger than the $\langle count \rangle$, give an empty result, and otherwise discard $\langle index \rangle - 1$ characters before returning the following one. The shift by -1 is obtained by inserting an empty brace group before the string in that case: that brace group also covers the case where the $\langle index \rangle$ is zero.

```

4308 \cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
4309 \cs_generate_variant:Nn \str_item:Nn { c }
4310 \cs_new:Npn \str_item:nn #1#2
4311 {
4312   \exp_args:Nf \tl_to_str:n
4313   {
4314     \exp_args:Nf \__str_item:nn
4315     { \__kernel_str_to_other:n {#1} } {#2}

```

```

4316     }
4317   }
4318   \cs_new:Npn \str_item_ignore_spaces:nn #1
4319   { \exp_args:No \__str_item:nn { \tl_to_str:n {#1} } }
4320   \cs_new:Npn \__str_item:nn #1#2
4321   {
4322     \exp_after:wN \__str_item:w
4323     \int_value:w \int_eval:n {#2} \exp_after:wN ;
4324     \int_value:w \__str_count:n {#1} ;
4325     #1 \q_stop
4326   }
4327   \cs_new:Npn \__str_item:w #1; #2;
4328   {
4329     \int_compare:nNnTF {#1} < 0
4330     {
4331       \int_compare:nNnTF {#1} < {-#2}
4332       { \use_none_delimit_by_q_stop:w }
4333       {
4334         \exp_after:wN \use_i_delimit_by_q_stop:nw
4335         \exp:w \exp_after:wN \__str_skip_exp_end:w
4336         \int_value:w \int_eval:n { #1 + #2 } ;
4337       }
4338     }
4339     {
4340       \int_compare:nNnTF {#1} > {#2}
4341       { \use_none_delimit_by_q_stop:w }
4342       {
4343         \exp_after:wN \use_i_delimit_by_q_stop:nw
4344         \exp:w \__str_skip_exp_end:w #1 ; { }
4345       }
4346     }
4347   }

```

(End definition for `\str_item:Nn` and others. These functions are documented on page 55.)

```

\__str_skip_exp_end:w
\__str_skip_loop:wNNNNNNNN
  \__str_skip_end:w
\__str_skip_end:NNNNNNNN

```

Removes $\max(\#1, 0)$ characters from the input stream, and then leaves `\exp_end:`. This should be expanded using `\exp:w`. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the `\if_case:w` construction leaves between 0 and 8 times the `\or:` control sequence, and those `\or:` become arguments of `__str_skip_end:NNNNNNNN`. If the number of characters to remove is 6, say, then there are two `\or:` left, and the 8 arguments of `__str_skip_end:NNNNNNNN` are the two `\or:`, and 6 characters from the input stream, exactly what we wanted to remove. Then close the `\if_case:w` conditional with `\fi:`, and stop the initial expansion with `\exp_end:` (see places where `__str_skip_exp_end:w` is called).

```

4348   \cs_new:Npn \__str_skip_exp_end:w #1;
4349   {
4350     \if_int_compare:w #1 > 8 \exp_stop_f:
4351     \exp_after:wN \__str_skip_loop:wNNNNNNNN
4352     \else:
4353     \exp_after:wN \__str_skip_end:w
4354     \int_value:w \int_eval:w
4355     \fi:
4356     #1 ;
4357   }

```

```

4358 \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
4359 {
4360     \exp_after:wN \__str_skip_exp_end:w
4361     \int_value:w \int_eval:n { #1 - 8 } ;
4362 }
4363 \cs_new:Npn \__str_skip_end:w #1 ;
4364 {
4365     \exp_after:wN \__str_skip_end:NNNNNNNN
4366     \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or: \or:
4367 }
4368 \cs_new:Npn \__str_skip_end:NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \exp_end: }

```

(End definition for __str_skip_exp_end:w and others.)

\str_range:Nnn Sanitize the string. Then evaluate the arguments. At this stage we also decrement the $\langle start\ index \rangle$, since our goal is to know how many characters should be removed. Then
\str_range:nnn limit the range to be non-negative and at most the length of the string (this avoids
\str_range_ignore_spaces:nnn needing to check for the end of the string when grabbing characters), shifting negative
__str_range:nnn numbers by the appropriate amount. Afterwards, skip characters, then keep some more,
__str_range:w and finally drop the end of the string.
__str_range:nnw

```

4369 \cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
4370 \cs_generate_variant:Nn \str_range:Nnn { c }
4371 \cs_new:Npn \str_range:nnn #1#2#3
4372 {
4373     \exp_args:Nf \tl_to_str:n
4374     {
4375         \exp_args:Nf \__str_range:nnn
4376         { \__kernel_str_to_other:n {#1} } {#2} {#3}
4377     }
4378 }
4379 \cs_new:Npn \str_range_ignore_spaces:nnn #1
4380 { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
4381 \cs_new:Npn \__str_range:nnn #1#2#3
4382 {
4383     \exp_after:wN \__str_range:w
4384     \int_value:w \__str_count:n {#1} \exp_after:wN ;
4385     \int_value:w \int_eval:n { (#2) - 1 } \exp_after:wN ;
4386     \int_value:w \int_eval:n {#3} ;
4387     #1 \q_stop
4388 }
4389 \cs_new:Npn \__str_range:w #1; #2; #3;
4390 {
4391     \exp_args:Nf \__str_range:nnw
4392     { \__str_range_normalize:nn {#2} {#1} }
4393     { \__str_range_normalize:nn {#3} {#1} }
4394 }
4395 \cs_new:Npn \__str_range:nnw #1#2
4396 {
4397     \exp_after:wN \__str_collect_delimit_by_q_stop:w
4398     \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
4399     \exp:w \__str_skip_exp_end:w #1 ;
4400 }

```

(End definition for \str_range:Nnn and others. These functions are documented on page 56.)

`__str_range_normalize:nn` This function converts an *⟨index⟩* argument into an explicit position in the string (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the *⟨index⟩* #1 and the string count #2. If #1 is negative, replace it by #1 + #2 + 1, then limit to the range [0, #2].

```

4401 \cs_new:Npn \__str_range_normalize:nn #1#2
4402 {
4403   \int_eval:n
4404   {
4405     \if_int_compare:w #1 < 0 \exp_stop_f:
4406     \if_int_compare:w #1 < -#2 \exp_stop_f:
4407       0
4408     \else:
4409       #1 + #2 + 1
4410     \fi:
4411   \else:
4412     \if_int_compare:w #1 < #2 \exp_stop_f:
4413       #1
4414     \else:
4415       #2
4416     \fi:
4417   \fi:
4418 }
4419 }
```

(End definition for `__str_range_normalize:nn`.)

`_str_collect_delimit_by_q_stop:w` Collects `max(#1,0)` characters, and removes everything else until `\q_stop`. This is somewhat similar to `__str_skip_exp_end:w`, but accepts integer expression arguments. This time we can only grab 7 characters at a time. At the end, we use an `\if_case:w` trick again, so that the 8 first arguments of `__str_collect_end:nnnnnnnnnw` are some `\or:`, followed by an `\fi:`, followed by #1 characters from the input stream. Simply leaving this in the input stream closes the conditional properly and the `\or:` disappear.

```

4420 \cs_new:Npn \__str_collect_delimit_by_q_stop:w #1;
4421 { \__str_collect_loop:wn #1 ; { } }
4422 \cs_new:Npn \__str_collect_loop:wn #1 ;
4423 {
4424   \if_int_compare:w #1 > 7 \exp_stop_f:
4425   \exp_after:wN \__str_collect_loop:wnNNNNNNN
4426   \else:
4427   \exp_after:wN \__str_collect_end:wn
4428   \fi:
4429   #1 ;
4430 }
4431 \cs_new:Npn \__str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
4432 {
4433   \exp_after:wN \__str_collect_loop:wn
4434   \int_value:w \int_eval:n { #1 - 7 } ;
4435   { #2 #3#4#5#6#7#8#9 }
4436 }
4437 \cs_new:Npn \__str_collect_end:wn #1 ;
4438 {
4439   \exp_after:wN \__str_collect_end:nnnnnnnnw
4440   \if_case:w \if_int_compare:w #1 > 0 \exp_stop_f:
```



```

4441     #1 \else: 0 \fi: \exp_stop_f:
4442     \or: \or: \or: \or: \or: \or: \fi:
4443 }
4444 \cs_new:Npn \__str_collect_end:nnnnnnnw #1#2#3#4#5#6#7#8 #9 \q_stop
4445 { #1#2#3#4#5#6#7#8 }

```

(End definition for `__str_collect_delimit_by_q_stop:w` and others.)

7.6 Counting characters

`\str_count_spaces:N` To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing `X⟨number⟩`, and that `⟨number⟩` is added to the sum of 9 that precedes, to adjust the result.

```

\str_count_spaces:c
\str_count_spaces:n
\__str_count_spaces_loop:w

4446 \cs_new:Npn \str_count_spaces:N
4447 { \exp_args:No \str_count_spaces:n }
4448 \cs_generate_variant:Nn \str_count_spaces:N { c }
4449 \cs_new:Npn \str_count_spaces:n #1
4450 {
4451     \int_eval:n
4452     {
4453         \exp_after:wN \__str_count_spaces_loop:w
4454         \tl_to_str:n {#1} ~
4455         X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
4456         \q_stop
4457     }
4458 }
4459 \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
4460 {
4461     \if_meaning:w X #9
4462     \use_i_delimit_by_q_stop:nw
4463     \fi:
4464     9 + \__str_count_spaces_loop:w
4465 }

```

(End definition for `\str_count_spaces:N`, `\str_count_spaces:n`, and `__str_count_spaces_loop:w`. These functions are documented on page 54.)

`\str_count:N` To count characters in a string we could first escape all spaces using `__kernel_str_to_other:n`, then pass the result to `\tl_count:n`. However, the escaping step would be quadratic in the number of characters in the string, and we can do better. Namely, sum the number of spaces (`\str_count_spaces:n`) and the result of `\tl_count:n`, which ignores spaces. Since strings tend to be longer than token lists, we use specialized functions to count characters ignoring spaces. Namely, `loop`, grabbing 9 non-space characters at each step, and end as soon as we reach one of the 9 trailing items. The internal function `__str_count:n`, used in `\str_item:nn` and `\str_range:nnn`, is similar to `\str_count_ignore_spaces:n` but expects its argument to already be a string or a string with spaces escaped.

```

\str_count:N
\str_count:c
\str_count:n
\str_count_ignore_spaces:n
\__str_count:n
\__str_count_aux:n
\__str_count_loop:NNNNNNNNN

4466 \cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
4467 \cs_generate_variant:Nn \str_count:N { c }
4468 \cs_new:Npn \str_count:n #1
4469 {
4470     \__str_count_aux:n

```

```

4471     {
4472       \str_count_spaces:n {#1}
4473       + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
4474     }
4475   }
4476 \cs_new:Npn \__str_count:n #1
4477 {
4478   \__str_count_aux:n
4479   { \__str_count_loop:NNNNNNNNN #1 }
4480 }
4481 \cs_new:Npn \str_count_ignore_spaces:n #1
4482 {
4483   \__str_count_aux:n
4484   { \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
4485 }
4486 \cs_new:Npn \__str_count_aux:n #1
4487 {
4488   \int_eval:n
4489   {
4490     #1
4491     { X 8 } { X 7 } { X 6 }
4492     { X 5 } { X 4 } { X 3 }
4493     { X 2 } { X 1 } { X 0 }
4494     \q_stop
4495   }
4496 }
4497 \cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
4498 {
4499   \if_meaning:w X #9
4500     \exp_after:wN \use_none_delimit_by_q_stop:w
4501   \fi:
4502   9 + \__str_count_loop:NNNNNNNNN
4503 }

```

(End definition for `\str_count:N` and others. These functions are documented on page 54.)

7.7 The first character in a string

`\str_head:N` The `_ignore_spaces` variant applies `\tl_to_str:n` then grabs the first item, thus skipping spaces. As usual, `\str_head:N` expands its argument and hands it to `\str_head:n`.
`\str_head:c` To circumvent the fact that \TeX skips spaces when grabbing undelimited macro parameters, `__str_head:w` takes an argument delimited by a space. If `#1` starts with a non-space character, `\use_i_delimit_by_q_stop:nw` leaves that in the input stream. On the other hand, if `#1` starts with a space, the `__str_head:w` takes an empty argument, and the single (initially braced) space in the definition of `__str_head:w` makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of `\str_head:n` gives an empty result after passing through `\use_i_delimit_by_q_stop:nw`.

```

4504 \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
4505 \cs_generate_variant:Nn \str_head:N { c }
4506 \cs_new:Npn \str_head:n #1
4507 {
4508   \exp_after:wN \__str_head:w

```

```

4509     \tl_to_str:n {#1}
4510     { { } } ~ \q_stop
4511 }
4512 \cs_new:Npn \__str_head:w #1 ~ %
4513 { \use_i_delimit_by_q_stop:nw #1 { ~ } }
4514 \cs_new:Npn \str_head_ignore_spaces:n #1
4515 {
4516     \exp_after:wN \use_i_delimit_by_q_stop:nw
4517     \tl_to_str:n {#1} { } \q_stop
4518 }

```

(End definition for `\str_head:N` and others. These functions are documented on page 55.)

`\str_tail:N` Getting the tail is a little bit more convoluted than the head of a string. We hit the front of the string with `\reverse_if:N \if_charcode:w \scan_stop:.` This removes the first character, and necessarily makes the test true, since the character cannot match `\scan_stop:.` The auxiliary function then inserts the required `\fi:` to close the conditional, and leaves the tail of the string in the input stream. The details are such that an empty string has an empty tail (this requires in particular that the end-marker `X` be unexpandable and not a control sequence). The `_ignore_spaces` is rather simpler: after converting the input to a string, `__str_tail_auxii:w` removes one undelimited argument and leaves everything else until an end-marker `\q_mark`. One can check that an empty (or blank) string yields an empty tail.

```

4519 \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
4520 \cs_generate_variant:Nn \str_tail:N { c }
4521 \cs_new:Npn \str_tail:n #1
4522 {
4523     \exp_after:wN \__str_tail_auxi:w
4524     \reverse_if:N \if_charcode:w
4525         \scan_stop: \tl_to_str:n {#1} X X \q_stop
4526 }
4527 \cs_new:Npn \__str_tail_auxi:w #1 X #2 \q_stop { \fi: #1 }
4528 \cs_new:Npn \str_tail_ignore_spaces:n #1
4529 {
4530     \exp_after:wN \__str_tail_auxii:w
4531     \tl_to_str:n {#1} \q_mark \q_mark \q_stop
4532 }
4533 \cs_new:Npn \__str_tail_auxii:w #1 #2 \q_mark #3 \q_stop { #2 }

```

(End definition for `\str_tail:N` and others. These functions are documented on page 55.)

7.8 String manipulation

`\str_fold_case:n` Case changing for programmatic reasons is done by first detokenizing input then doing a simple loop that only has to worry about spaces and everything else. The output is detokenized to allow data sharing with text-based case changing.

```

4534 \cs_new:Npn \str_fold_case:n #1 { \__str_change_case:nn {#1} { fold } }
4535 \cs_new:Npn \str_lower_case:n #1 { \__str_change_case:nn {#1} { lower } }
4536 \cs_new:Npn \str_upper_case:n #1 { \__str_change_case:nn {#1} { upper } }
4537 \cs_generate_variant:Nn \str_fold_case:n { V }
4538 \cs_generate_variant:Nn \str_lower_case:n { f }
4539 \cs_generate_variant:Nn \str_upper_case:n { f }
4540 \cs_new:Npn \__str_change_case:nn #1

```

```

\__str_change_case:nn
\__str_change_case_aux:nn
\__str_change_case_result:n
\__str_change_case_output:nw
\__str_change_case_output:fw
\__str_change_case_end:nw
\__str_change_case_loop:nw
\__str_change_case_space:n
\__str_change_case_char:nN

```

```

4541 {
4542   \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
4543   { \tl_to_str:n {#1} }
4544 }
4545 \cs_new:Npn \__str_change_case_aux:nn #1#2
4546 {
4547   \__str_change_case_loop:nw {#2} #1 \q_recursion_tail \q_recursion_stop
4548   \__str_change_case_result:n { }
4549 }
4550 \cs_new:Npn \__str_change_case_output:nw #1#2 \__str_change_case_result:n #3
4551 { #2 \__str_change_case_result:n { #3 #1 } }
4552 \cs_generate_variant:Nn \__str_change_case_output:nw { f }
4553 \cs_new:Npn \__str_change_case_end:wn #1 \__str_change_case_result:n #2
4554 { \tl_to_str:n {#2} }
4555 \cs_new:Npn \__str_change_case_loop:nw #1#2 \q_recursion_stop
4556 {
4557   \tl_if_head_is_space:nTF {#2}
4558   { \__str_change_case_space:n }
4559   { \__str_change_case_char:nN }
4560   {#1} #2 \q_recursion_stop
4561 }
4562 \exp_last_unbraced:NNNNo
4563 \cs_new:Npn \__str_change_case_space:n #1 \c_space_tl
4564 {
4565   \__str_change_case_output:nw { ~ }
4566   \__str_change_case_loop:nw {#1}
4567 }
4568 \cs_new:Npn \__str_change_case_char:nN #1#2
4569 {
4570   \quark_if_recursion_tail_stop_do:Nn #2
4571   { \__str_change_case_end:wn }
4572   \__str_change_case_output:fw
4573   { \use:c { char_ #1 _case:N } #2 }
4574   \__str_change_case_loop:nw {#1}
4575 }

```

(End definition for `\str_fold_case:n` and others. These functions are documented on page 58.)

<code>\c_ampersand_str</code>	For all of those strings, use <code>\cs_to_str:N</code> to get characters with the correct category
<code>\c_atsign_str</code>	code without worries
<code>\c_backslash_str</code>	4576 <code>\str_const:Nx \c_ampersand_str { \cs_to_str:N \& }</code>
<code>\c_left_brace_str</code>	4577 <code>\str_const:Nx \c_atsign_str { \cs_to_str:N \@ }</code>
<code>\c_right_brace_str</code>	4578 <code>\str_const:Nx \c_backslash_str { \cs_to_str:N \\ }</code>
<code>\c_circumflex_str</code>	4579 <code>\str_const:Nx \c_left_brace_str { \cs_to_str:N \{ }</code>
<code>\c_colon_str</code>	4580 <code>\str_const:Nx \c_right_brace_str { \cs_to_str:N \} }</code>
<code>\c_dollar_str</code>	4581 <code>\str_const:Nx \c_circumflex_str { \cs_to_str:N ^ }</code>
<code>\c_hash_str</code>	4582 <code>\str_const:Nx \c_colon_str { \cs_to_str:N : }</code>
<code>\c_percent_str</code>	4583 <code>\str_const:Nx \c_dollar_str { \cs_to_str:N \$ }</code>
<code>\c_tilde_str</code>	4584 <code>\str_const:Nx \c_hash_str { \cs_to_str:N # }</code>
<code>\c_underscore_str</code>	4585 <code>\str_const:Nx \c_percent_str { \cs_to_str:N % }</code>
	4586 <code>\str_const:Nx \c_tilde_str { \cs_to_str:N ~ }</code>
	4587 <code>\str_const:Nx \c_underscore_str { \cs_to_str:N _ }</code>

(End definition for `\c_ampersand_str` and others. These variables are documented on page 59.)

`\l_tmpa_str` Scratch strings.

```
\l_tmpb_str 4588 \str_new:N \l_tmpa_str
\g_tmpa_str 4589 \str_new:N \l_tmpb_str
\g_tmpb_str 4590 \str_new:N \g_tmpa_str
4591 \str_new:N \g_tmpb_str
```

(End definition for `\l_tmpa_str` and others. These variables are documented on page 59.)

7.9 Viewing strings

`\str_show:n` Displays a string on the terminal.

```
\str_show:N 4592 \cs_new_eq:NN \str_show:n \tl_show:n
\str_show:c 4593 \cs_new_eq:NN \str_show:N \tl_show:N
4594 \cs_generate_variant:Nn \str_show:N { c }
```

(End definition for `\str_show:n` and `\str_show:N`. These functions are documented on page 58.)

```
4595 </initex | package>
```

8 l3quark implementation

The following test files are used for this code: `m3quark001.lvt`.

```
4596 <*initex | package>
```

8.1 Quarks

```
4597 <@@=quark>
```

`\quark_new:N` Allocate a new quark.

```
4598 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN q #1 } { }
4599 \cs_new_protected:Npn \quark_new:N #1
4600 {
4601   \__kernel_chk_if_free_cs:N #1
4602   \cs_gset_nopar:Npn #1 {#1}
4603 }
```

(End definition for `\quark_new:N`. This function is documented on page 60.)

`\q_nil` Some “public” quarks. `\q_stop` is an “end of argument” marker, `\q_nil` is a empty value and `\q_no_value` marks an empty argument.

```
\q_mark 4604 \quark_new:N \q_nil
\q_no_value 4605 \quark_new:N \q_mark
\q_stop 4606 \quark_new:N \q_no_value
4607 \quark_new:N \q_stop
```

(End definition for `\q_nil` and others. These variables are documented on page 61.)

`\q_recursion_tail` Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. `\q_recursion_stop` is placed directly after the list.

```
4608 \quark_new:N \q_recursion_tail
4609 \quark_new:N \q_recursion_stop
```

(End definition for `\q_recursion_tail` and `\q_recursion_stop`. These variables are documented on page 61.)

`\quark_if_recursion_tail_stop:N`
`\quark_if_recursion_tail_stop_do:Nn`

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```

4610 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
4611 {
4612   \if_meaning:w \q_recursion_tail #1
4613   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
4614   \fi:
4615 }
4616 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
4617 {
4618   \if_meaning:w \q_recursion_tail #1
4619   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
4620   \else:
4621     \exp_after:wN \use_none:n
4622   \fi:
4623 }
```

(End definition for `\quark_if_recursion_tail_stop:N` and `\quark_if_recursion_tail_stop_do:Nn`. These functions are documented on page 62.)

`\quark_if_recursion_tail_stop:n`
`\quark_if_recursion_tail_stop:o`
`\quark_if_recursion_tail_stop_do:nn`
`\quark_if_recursion_tail_stop_do:nn`
`__quark_if_recursion_tail:w`

See `\quark_if_nil:nTF` for the details. Expanding `__quark_if_recursion_tail:w` once in front of the tokens chosen here gives an empty result if and only if #1 is exactly `\q_recursion_tail`.

```

4624 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
4625 {
4626   \tl_if_empty:oTF
4627   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
4628   { \use_none_delimit_by_q_recursion_stop:w }
4629   { }
4630 }
4631 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
4632 {
4633   \tl_if_empty:oTF
4634   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
4635   { \use_i_delimit_by_q_recursion_stop:nw }
4636   { \use_none:n }
4637 }
4638 \cs_new:Npn \__quark_if_recursion_tail:w
4639   #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
4640 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
4641 \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }
```

(End definition for `\quark_if_recursion_tail_stop:n`, `\quark_if_recursion_tail_stop_do:nn`, and `__quark_if_recursion_tail:w`. These functions are documented on page 62.)

`\quark_if_recursion_tail_break:NN`
`\quark_if_recursion_tail_break:nN`

Analogues of the `\quark_if_recursion_tail_stop...` functions. Break the mapping using #2.

```

4642 \cs_new:Npn \quark_if_recursion_tail_break:NN #1#2
4643 {
4644   \if_meaning:w \q_recursion_tail #1
4645   \exp_after:wN #2
4646   \fi:
4647 }
4648 \cs_new:Npn \quark_if_recursion_tail_break:nN #1#2
4649 {
4650   \tl_if_empty:oT
4651   { \_quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??? }
4652   {#2}
4653 }

```

(End definition for \quark_if_recursion_tail_break:NN and \quark_if_recursion_tail_break:nN. These functions are documented on page 62.)

```

\quark_if_nil_p:N Here we test if we found a special quark as the first argument. We better start with
\quark_if_nil:NTF \q_no_value as the first argument since the whole thing may otherwise loop if #1 is
\quark_if_no_value_p:N wrongly given a string like aabc instead of a single token.7
\quark_if_no_value_p:c 4654 \prg_new_conditional:Npnn \quark_if_nil:N #1 { p , T , F , TF }
\quark_if_no_value:NTF 4655 {
\quark_if_no_value:cTF 4656   \if_meaning:w \q_nil #1
4657   \prg_return_true:
4658   \else:
4659   \prg_return_false:
4660   \fi:
4661 }
4662 \prg_new_conditional:Npnn \quark_if_no_value:N #1 { p , T , F , TF }
4663 {
4664   \if_meaning:w \q_no_value #1
4665   \prg_return_true:
4666   \else:
4667   \prg_return_false:
4668   \fi:
4669 }
4670 \prg_generate_conditional_variant:Nnn \quark_if_no_value:N
4671 { c } { p , T , F , TF }

```

(End definition for \quark_if_nil:NTF and \quark_if_no_value:NTF. These functions are documented on page 61.)

```

\quark_if_nil_p:n Let us explain \quark_if_nil:n(TF). Expanding \_quark_if_nil:w once is safe
\quark_if_nil_p:V thanks to the trailing \q_nil ??!. The result of expanding once is empty if and only
\quark_if_nil_p:o if both delimited arguments #1 and #2 are empty and #3 is delimited by the last to-
\quark_if_nil:nTF kens ?!. Thanks to the leading {}, the argument #1 is empty if and only if the argument
\quark_if_nil:VTF of \quark_if_nil:n starts with \q_nil. The argument #2 is empty if and only if this
\quark_if_nil:oTF \q_nil is followed immediately by ? or by {}?, coming either from the trailing tokens in
\quark_if_no_value_p:n the definition of \quark_if_nil:n, or from its argument. In the first case, \_quark_
\quark_if_no_value:NTF if_nil:w is followed by {} \q_nil {}? ! \q_nil ??!, hence #3 is delimited by the final ?!,
\_quark_if_nil:w and the test returns true as wanted. In the second case, the result is not empty since
\_quark_if_no_value:w the first ?! in the definition of \quark_if_nil:n stop #3. The auxiliary here is the same
\_quark_if_empty_return:o as \_tl_if_empty_return:o, with the same comments applying.

```

⁷It may still loop in special circumstances however!

```

4672 \prg_new_conditional:Npnn \quark_if_nil:n #1 { p, T , F , TF }
4673 {
4674   \__quark_if_empty_return:o
4675   { \__quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
4676 }
4677 \cs_new:Npn \__quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
4678 \prg_new_conditional:Npnn \quark_if_no_value:n #1 { p, T , F , TF }
4679 {
4680   \__quark_if_empty_return:o
4681   { \__quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
4682 }
4683 \cs_new:Npn \__quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
4684 \prg_generate_conditional_variant:Nnn \quark_if_nil:n
4685 { V , o } { p , TF , T , F }
4686 \cs_new:Npn \__quark_if_empty_return:o #1
4687 {
4688   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
4689   \__kernel_tl_to_str:w \exp_after:wN {#1} \q_nil
4690   \prg_return_true:
4691   \else:
4692     \prg_return_false:
4693   \fi:
4694 }

```

(End definition for \quark_if_nil:nTF and others. These functions are documented on page 61.)

8.2 Scan marks

```

4695 <@@=scan>

```

\g__scan_marks_tl The list of all scan marks currently declared.

```

4696 \tl_new:N \g__scan_marks_tl

```

(End definition for \g__scan_marks_tl.)

\scan_new:N Check whether the variable is already a scan mark, then declare it to be equal to \scan_stop: globally.

```

4697 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN s #1 } { }
4698 \cs_new_protected:Npn \scan_new:N #1
4699 {
4700   \tl_if_in:NnTF \g__scan_marks_tl { #1 }
4701   {
4702     \__kernel_msg_error:nxx { kernel } { scanmark-already-defined }
4703     { \token_to_str:N #1 }
4704   }
4705   {
4706     \tl_gput_right:Nn \g__scan_marks_tl {#1}
4707     \cs_new_eq:NN #1 \scan_stop:
4708   }
4709 }

```

(End definition for \scan_new:N. This function is documented on page 63.)

\s_stop We only declare one scan mark here, more can be defined by specific modules.

```

4710 \scan_new:N \s_stop

```


(End definition for `\s_stop`. This variable is documented on page 63.)

`\use_none_delimit_by_s_stop:w` Similar to `\use_none_delimit_by_q_stop:w`.

```
4711 \cs_new:Npn \use_none_delimit_by_s_stop:w #1 \s_stop { }
```

(End definition for `\use_none_delimit_by_s_stop:w`. This function is documented on page 64.)

```
4712 </initex | package>
```

9 l3seq implementation

The following test files are used for this code: `m3seq002,m3seq003`.

```
4713 (*initex | package)
```

```
4714 <@@=seq>
```

A sequence is a control sequence whose top-level expansion is of the form “`\s__seq __seq_item:n {<item1>} ... __seq_item:n {<itemn>}`”, with a leading scan mark followed by n items of the same form. An earlier implementation used the structure “`\seq_elt:w <item1> \seq_elt_end: ... \seq_elt:w <itemn> \seq_elt_end:`”. This allowed rapid searching using a delimited function, but was not suitable for items containing `{`, `}` and `#` tokens, and also lead to the loss of surrounding braces around items

`__seq_item:n` ★ `__seq_item:n {<item>}`

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

`__seq_push_item_def:n` `__seq_push_item_def:n {<code>}`
`__seq_push_item_def:x`

Saves the definition of `__seq_item:n` and redefines it to accept one parameter and expand to `<code>`. This function should always be balanced by use of `__seq_pop_item_def:`.

`__seq_pop_item_def:` `__seq_pop_item_def:`

Restores the definition of `__seq_item:n` most recently saved by `__seq_push_item_def:n`. This function should always be used in a balanced pair with `__seq_push_item_def:n`.

`\s__seq` This private scan mark.

```
4715 \scan_new:N \s__seq
```

(End definition for `\s__seq`.)

`__seq_item:n` The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

```
4716 \cs_new:Npn \__seq_item:n
4717 {
4718   \__kernel_msg_expandable_error:nn { kernel } { misused-sequence }
4719   \use_none:n
4720 }
```

(End definition for `__seq_item:n`.)

`\l__seq_internal_a_tl` Scratch space for various internal uses.
`\l__seq_internal_b_tl` 4721 `\tl_new:N \l__seq_internal_a_tl`
4722 `\tl_new:N \l__seq_internal_b_tl`
(End definition for `\l__seq_internal_a_tl` and `\l__seq_internal_b_tl`.)

`__seq_tmp:w` Scratch function for internal use.
4723 `\cs_new_eq:NN __seq_tmp:w ?`
(End definition for `__seq_tmp:w`.)

`\c_empty_seq` A sequence with no item, following the structure mentioned above.
4724 `\tl_const:Nn \c_empty_seq { \s__seq }`
(End definition for `\c_empty_seq`. This variable is documented on page 74.)

9.1 Allocation and initialisation

`\seq_new:N` Sequences are initialized to `\c_empty_seq`.
`\seq_new:c` 4725 `\cs_new_protected:Npn \seq_new:N #1`
4726 `{`
4727 `__kernel_chk_if_free_cs:N #1`
4728 `\cs_gset_eq:NN #1 \c_empty_seq`
4729 `}`
4730 `\cs_generate_variant:Nn \seq_new:N { c }`
(End definition for `\seq_new:N`. This function is documented on page 65.)

`\seq_clear:N` Clearing a sequence is similar to setting it equal to the empty one.
`\seq_clear:c` 4731 `\cs_new_protected:Npn \seq_clear:N #1`
`\seq_gclear:N` 4732 `{ \seq_set_eq:NN #1 \c_empty_seq }`
`\seq_gclear:c` 4733 `\cs_generate_variant:Nn \seq_clear:N { c }`
4734 `\cs_new_protected:Npn \seq_gclear:N #1`
4735 `{ \seq_gset_eq:NN #1 \c_empty_seq }`
4736 `\cs_generate_variant:Nn \seq_gclear:N { c }`
(End definition for `\seq_clear:N` and `\seq_gclear:N`. These functions are documented on page 65.)

`\seq_clear_new:N` Once again we copy code from the token list functions.
`\seq_clear_new:c` 4737 `\cs_new_protected:Npn \seq_clear_new:N #1`
`\seq_gclear_new:N` 4738 `{ \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }`
`\seq_gclear_new:c` 4739 `\cs_generate_variant:Nn \seq_clear_new:N { c }`
4740 `\cs_new_protected:Npn \seq_gclear_new:N #1`
4741 `{ \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }`
4742 `\cs_generate_variant:Nn \seq_gclear_new:N { c }`
(End definition for `\seq_clear_new:N` and `\seq_gclear_new:N`. These functions are documented on page 65.)

\seq_set_eq:NN Copying a sequence is the same as copying the underlying token list.

```

\seq_set_eq:cN 4743 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
\seq_set_eq:Nc 4744 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
\seq_set_eq:cc 4745 \cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN
\seq_gset_eq:NN 4746 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
\seq_gset_eq:cN 4747 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
\seq_gset_eq:Nc 4748 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
\seq_gset_eq:cc 4749 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
4750 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc

```

(End definition for `\seq_set_eq:NN` and `\seq_gset_eq:NN`. These functions are documented on page 65.)

\seq_set_from_clist:NN Setting a sequence from a comma-separated list is done using a simple mapping.

```

\seq_set_from_clist:cN 4751 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
\seq_set_from_clist:Nc 4752 {
\seq_set_from_clist:cc 4753   \tl_set:Nx #1
\seq_set_from_clist:Nn 4754   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
\seq_set_from_clist:cn 4755 }
\seq_gset_from_clist:NN 4756 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
\seq_gset_from_clist:cN 4757 {
\seq_gset_from_clist:Nc 4758   \tl_set:Nx #1
\seq_gset_from_clist:cc 4759   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
4760 }
\seq_gset_from_clist:Nn 4761 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
\seq_gset_from_clist:cn 4762 {
4763   \tl_gset:Nx #1
4764   { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
4765 }
4766 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
4767 {
4768   \tl_gset:Nx #1
4769   { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
4770 }
4771 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
4772 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
4773 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c , cc }
4774 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
4775 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
4776 \cs_generate_variant:Nn \seq_gset_from_clist:Nn { c , cc }

```

(End definition for `\seq_set_from_clist:NN` and others. These functions are documented on page 65.)

\seq_set_split:Nnn When the separator is empty, everything is very simple, just map `__seq_wrap_item:n` through the items of the last argument. For non-trivial separators, the goal is to split a given token list at the marker, strip spaces from each item, and remove one set of outer braces if after removing leading and trailing spaces the item is enclosed within braces. After `\tl_replace_all:Nnn`, the token list `\l__seq_internal_a_tl` is a repetition of the pattern `__seq_set_split_auxi:w \prg_do_nothing: <item with spaces> __seq_set_split_end:.` Then, x-expansion causes `__seq_set_split_auxi:w` to trim spaces, and leaves its result as `__seq_set_split_auxii:w <trimmed item> __seq_set_split_end:.` This is then converted to the `l3seq` internal structure by another x-expansion. In the first step, we insert `\prg_do_nothing:` to avoid losing braces too early:

that would cause space trimming to act within those lost braces. The second step is solely there to strip braces which are outermost after space trimming.

```

4777 \cs_new_protected:Npn \seq_set_split:Nnn
4778 { \__seq_set_split:NNnn \tl_set:Nx }
4779 \cs_new_protected:Npn \seq_gset_split:Nnn
4780 { \__seq_set_split:NNnn \tl_gset:Nx }
4781 \cs_new_protected:Npn \__seq_set_split:NNnn #1#2#3#4
4782 {
4783   \tl_if_empty:nTF {#3}
4784   {
4785     \tl_set:Nn \l__seq_internal_a_tl
4786     { \tl_map_function:nN {#4} \__seq_wrap_item:n }
4787   }
4788   {
4789     \tl_set:Nn \l__seq_internal_a_tl
4790     {
4791       \__seq_set_split_auxi:w \prg_do_nothing:
4792       #4
4793       \__seq_set_split_end:
4794     }
4795     \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
4796     {
4797       \__seq_set_split_end:
4798       \__seq_set_split_auxi:w \prg_do_nothing:
4799     }
4800     \tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
4801   }
4802   #1 #2 { \s__seq \l__seq_internal_a_tl }
4803 }
4804 \cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
4805 {
4806   \exp_not:N \__seq_set_split_auxii:w
4807   \exp_args:No \tl_trim_spaces:n {#1}
4808   \exp_not:N \__seq_set_split_end:
4809 }
4810 \cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
4811 { \__seq_wrap_item:n {#1} }
4812 \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
4813 \cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }

```

(End definition for `\seq_set_split:Nnn` and others. These functions are documented on page 66.)

`\seq_concat:NNN` When concatenating sequences, one must remove the leading `\s__seq` of the second sequence. The result starts with `\s__seq` (of the first sequence), which stops `f`-expansion.

`\seq_concat:ccc`

`\seq_gconcat:NNN`

`\seq_gconcat:ccc`

```

4814 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
4815 { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
4816 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
4817 { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
4818 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
4819 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_gconcat:NNN`. These functions are documented on page 66.)

```

\seq_if_exist_p:N Copies of the cs functions defined in l3basics.
\seq_if_exist_p:c 4820 \prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N
\seq_if_exist:NTF 4821 { TF , T , F , p }
\seq_if_exist:cTF 4822 \prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c
4823 { TF , T , F , p }

```

(End definition for \seq_if_exist:N~~TF~~. This function is documented on page 66.)

9.2 Appending data to either end

```

\seq_put_left:Nn When adding to the left of a sequence, remove \s__seq. This is done by \__seq_put_
\seq_put_left:NV left_aux:w, which also stops f-expansion.
\seq_put_left:Nv 4824 \cs_new_protected:Npn \seq_put_left:Nn #1#2
\seq_put_left:No 4825 {
\seq_put_left:Nx 4826   \tl_set:Nx #1
\seq_put_left:cn 4827   {
\seq_put_left:cV 4828     \exp_not:n { \s__seq \__seq_item:n {#2} }
\seq_put_left:cv 4829     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
\seq_put_left:co 4830   }
\seq_put_left:cx 4831 }
\seq_gput_left:Nn 4832 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
\seq_gput_left:NV 4833 {
\seq_gput_left:Nv 4834   \tl_gset:Nx #1
\seq_gput_left:No 4835   {
\seq_gput_left:Nx 4836     \exp_not:n { \s__seq \__seq_item:n {#2} }
\seq_gput_left:cn 4837     \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
\seq_gput_left:cV 4838   }
\seq_gput_left:cv 4839 }
\seq_gput_left:co 4840 \cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
\seq_gput_left:cx 4841 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
\__seq_put_left_aux:w 4842 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
4843 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
4844 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }

```

(End definition for \seq_put_left:Nn, \seq_gput_left:Nn, and __seq_put_left_aux:w. These functions are documented on page 66.)

```

\seq_put_right:Nn Since there is no trailing marker, adding an item to the right of a sequence simply means
\seq_put_right:NV wrapping it in \__seq_item:n.
\seq_put_right:Nv 4845 \cs_new_protected:Npn \seq_put_right:Nn #1#2
\seq_put_right:No 4846 { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
\seq_put_right:Nx 4847 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
\seq_put_right:cn 4848 { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
\seq_put_right:cV 4849 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
\seq_put_right:cv 4850 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
\seq_put_right:co 4851 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
\seq_put_right:cx 4852 \cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , co , cx }
\seq_gput_right:Nn (End definition for \seq_put_right:Nn and \seq_gput_right:Nn. These functions are documented on
\seq_gput_right:NV page 66.)
\seq_gput_right:Nv
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:cv
\seq_gput_right:co
\seq_gput_right:cx

```

9.3 Modifying sequences

`__seq_wrap_item:n` This function converts its argument to a proper sequence item in an x-expansion context.

```
4853 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }
```

(End definition for __seq_wrap_item:n.)

`\l__seq_remove_seq` An internal sequence for the removal routines.

```
4854 \seq_new:N \l__seq_remove_seq
```

(End definition for \l__seq_remove_seq.)

`\seq_remove_duplicates:N` Removing duplicates means making a new list then copying it.

```
\seq_remove_duplicates:c 4855 \cs_new_protected:Npn \seq_remove_duplicates:N
\seq_gremove_duplicates:N 4856 { \__seq_remove_duplicates:NN \seq_set_eq:NN }
\seq_gremove_duplicates:c 4857 \cs_new_protected:Npn \seq_gremove_duplicates:N
\__seq_remove_duplicates:NN 4858 { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
4859 \cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
4860 {
4861   \seq_clear:N \l__seq_remove_seq
4862   \seq_map_inline:Nn #2
4863   {
4864     \seq_if_in:NnF \l__seq_remove_seq {##1}
4865     { \seq_put_right:Nn \l__seq_remove_seq {##1} }
4866   }
4867   #1 #2 \l__seq_remove_seq
4868 }
4869 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
4870 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }
```

(End definition for \seq_remove_duplicates:N, \seq_gremove_duplicates:N, and __seq_remove_duplicates:NN. These functions are documented on page 69.)

`\seq_remove_all:Nn` The idea of the code here is to avoid a relatively expensive addition of items one at a time
`\seq_remove_all:cn` to an intermediate sequence. The approach taken is therefore similar to that in `__seq_pop_right:NNN`, using a “flexible” x-type expansion to do most of the work. As `\tl_if_eq:nnT` is not expandable, a two-part strategy is needed. First, the x-type expansion uses `\str_if_eq:nnT` to find potential matches. If one is found, the expansion is halted and the necessary set up takes place to use the `\tl_if_eq:nnT` test. The x-type is started again, including all of the items copied already. This happens repeatedly until the entire sequence has been scanned. The code is set up to avoid needing and intermediate scratch list: the lead-off x-type expansion (`#1 #2 {#2}`) ensures that nothing is lost.

```
4871 \cs_new_protected:Npn \seq_remove_all:Nn
4872 { \__seq_remove_all_aux:NNn \tl_set:Nx }
4873 \cs_new_protected:Npn \seq_gremove_all:Nn
4874 { \__seq_remove_all_aux:NNn \tl_gset:Nx }
4875 \cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
4876 {
4877   \__seq_push_item_def:n
4878   {
4879     \str_if_eq:nnT {##1} {#3}
4880     {
4881       \if_false: { \fi: }
4882       \tl_set:Nn \l__seq_internal_b_tl {##1}
```

```

4883         #1 #2
4884         { \if_false: } \fi:
4885         \exp_not:o {#2}
4886         \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
4887         { \use_none:nn }
4888     }
4889     \__seq_wrap_item:n {##1}
4890 }
4891 \tl_set:Nn \l__seq_internal_a_tl {#3}
4892 #1 #2 {#2}
4893 \__seq_pop_item_def:
4894 }
4895 \cs_generate_variant:Nn \seq_remove_all:Nn { c }
4896 \cs_generate_variant:Nn \seq_gremove_all:Nn { c }

```

(End definition for `\seq_remove_all:Nn`, `\seq_gremove_all:Nn`, and `__seq_remove_all_aux:NNn`. These functions are documented on page 69.)

```

\seq_reverse:N Previously, \seq_reverse:N was coded by collecting the items in reverse order after an
\seq_reverse:c \exp_stop_f: marker.
\seq_greverse:N
\seq_greverse:c
\__seq_reverse:NN
\__seq_reverse_item:nwn
\cs_new_protected:Npn \seq_reverse:N #1
{
  \cs_set_eq:NN \@@_item:n \@@_reverse_item:nw
  \tl_set:Nf #2 { #2 \exp_stop_f: }
}
\cs_new:Npn \@@_reverse_item:nw #1 #2 \exp_stop_f:
{
  #2 \exp_stop_f:
  \@@_item:n {#1}
}

```

At first, this seems optimal, since we can forget about each item as soon as it is placed after `\exp_stop_f:`. Unfortunately, \TeX 's usual tail recursion does not take place in this case: since the following `__seq_reverse_item:nw` only reads tokens until `\exp_stop_f:`, and never reads the `\@@_item:n {#1}` left by the previous call, \TeX cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus only flushed after all the `__seq_reverse_item:nw` are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. \TeX can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of `\exp_not:n`. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```

4897 \cs_new_protected:Npn \seq_reverse:N
4898 { \__seq_reverse:NN \tl_set:Nx }
4899 \cs_new_protected:Npn \seq_greverse:N
4900 { \__seq_reverse:NN \tl_gset:Nx }
4901 \cs_new_protected:Npn \__seq_reverse:NN #1 #2
4902 {
4903   \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
4904   \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
4905   #1 #2 { #2 \exp_not:n { } }
4906   \cs_set_eq:NN \__seq_item:n \__seq_tmp:w

```

```

4907 }
4908 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
4909 {
4910     #2
4911     \exp_not:n { \__seq_item:n {#1} #3 }
4912 }
4913 \cs_generate_variant:Nn \seq_reverse:N { c }
4914 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for `\seq_reverse:N` and others. These functions are documented on page 69.)

`\seq_sort:Nn` Implemented in `l3sort`.

`\seq_sort:cn`

`\seq_gsort:Nn` (End definition for `\seq_sort:Nn` and `\seq_gsort:Nn`. These functions are documented on page 69.)

`\seq_gsort:cn`

9.4 Sequence conditionals

`\seq_if_empty_p:N` Similar to token lists, we compare with the empty sequence.

```

\seq_if_empty_p:c 4915 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
\seq_if_empty:NTF 4916 {
\seq_if_empty:cTF 4917     \if_meaning:w #1 \c_empty_seq
4918     \prg_return_true:
4919     \else:
4920     \prg_return_false:
4921     \fi:
4922 }
4923 \prg_generate_conditional_variant:Nnn \seq_if_empty:N
4924 { c } { p , T , F , TF }

```

(End definition for `\seq_if_empty:N`. This function is documented on page 69.)

`\seq_if_in:NnTF` The approach here is to define `__seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\group_end: \prg_return_true:` is inserted after skipping over the rest of the recursion. On the other hand, if there is no match then the loop breaks, returning `\prg_return_false:`. Everything is inside a group so that `__seq_item:n` is preserved in nested situations.

```

\seq_if_in:NnTF 4925 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
\seq_if_in:NvTF 4926 { T , F , TF }
\seq_if_in:NvTF 4927 {
\seq_if_in:NoTF 4928     \group_begin:
\seq_if_in:NxTF 4929     \tl_set:Nn \l__seq_internal_a_tl {#2}
\seq_if_in:cnTF 4930     \cs_set_protected:Npn \__seq_item:n ##1
\seq_if_in:cVTF 4931     {
\seq_if_in:cvTF 4932         \tl_set:Nn \l__seq_internal_b_tl {##1}
\seq_if_in:coTF 4933         \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
\seq_if_in:cxTF 4934         \exp_after:wN \__seq_if_in:
4935         \fi:
4936     }
4937     #1
4938     \group_end:
4939     \prg_return_false:
4940     \prg_break_point:
4941 }
4942 \cs_new:Npn \__seq_if_in:

```



```

4943 { \prg_break:n { \group_end: \prg_return_true: } }
4944 \prg_generate_conditional_variant:Nnn \seq_if_in:Nn
4945 { NV , Nv , No , Nx , c , cV , cv , co , cx } { T , F , TF }

```

(End definition for `\seq_if_in:NnTF` and `__seq_if_in:..` This function is documented on page 69.)

9.5 Recovering data from sequences

`__seq_pop:NNNN` The two pop functions share their emptiness tests. We also use a common emptiness test
`__seq_pop_TF:NNNN` for all branching get and pop functions.

```

4946 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
4947 {
4948   \if_meaning:w #3 \c_empty_seq
4949   \tl_set:Nn #4 { \q_no_value }
4950   \else:
4951     #1#2#3#4
4952   \fi:
4953 }
4954 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
4955 {
4956   \if_meaning:w #3 \c_empty_seq
4957   % \tl_set:Nn #4 { \q_no_value }
4958   \prg_return_false:
4959   \else:
4960     #1#2#3#4
4961   \prg_return_true:
4962   \fi:
4963 }

```

(End definition for `__seq_pop:NNNN` and `__seq_pop_TF:NNNN`.)

`\seq_get_left:NN` Getting an item from the left of a sequence is pretty easy: just trim off the first item
`\seq_get_left:cN` after `__seq_item:n` at the start. We append a `\q_no_value` item to cover the case of
`__seq_get_left:wnw` an empty sequence

```

4964 \cs_new_protected:Npn \seq_get_left:NN #1#2
4965 {
4966   \tl_set:Nx #2
4967   {
4968     \exp_after:wN \__seq_get_left:wnw
4969     #1 \__seq_item:n { \q_no_value } \q_stop
4970   }
4971 }
4972 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \q_stop
4973 { \exp_not:n {#2} }
4974 \cs_generate_variant:Nn \seq_get_left:NN { c }

```

(End definition for `\seq_get_left:NN` and `__seq_get_left:wnw`. This function is documented on page 66.)

`\seq_pop_left:NN` The approach to popping an item is pretty similar to that to get an item, with the only
`\seq_pop_left:cN` difference being that the sequence itself has to be redefined. This makes it more sensible
`\seq_gpop_left:NN` to use an auxiliary function for the local and global cases.
`\seq_gpop_left:cN`
`__seq_pop_left:NNN` 4975 \cs_new_protected:Npn \seq_pop_left:NN
`__seq_pop_left:wnwNNN` 4976 { __seq_pop:NNNN __seq_pop_left:NNN \tl_set:Nn }

```

4977 \cs_new_protected:Npn \seq_gpop_left:NN
4978 { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
4979 \cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
4980 { \exp_after:wN \__seq_pop_left:wnwNNN #2 \q_stop #1#2#3 }
4981 \cs_new_protected:Npn \__seq_pop_left:wnwNNN
4982 #1 \__seq_item:n #2#3 \q_stop #4#5#6
4983 {
4984 #4 #5 { #1 #3 }
4985 \tl_set:Nn #6 {#2}
4986 }
4987 \cs_generate_variant:Nn \seq_pop_left:NN { c }
4988 \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for `\seq_pop_left:NN` and others. These functions are documented on page 67.)

`\seq_get_right:NN` First remove `\s__seq` and prepend `\q_no_value`. The first argument of `__seq_get_right_loop:nw` is the last item found, and the second argument is empty until the end of the loop, where it is code that applies `\exp_not:n` to the last item and ends the loop.

```

\seq_get_right:cN
\__seq_get_right_loop:nw
\__seq_get_right_end:NnN
4989 \cs_new_protected:Npn \seq_get_right:NN #1#2
4990 {
4991 \tl_set:Nx #2
4992 {
4993 \exp_after:wN \use_i_ii:nnn
4994 \exp_after:wN \__seq_get_right_loop:nw
4995 \exp_after:wN \q_no_value
4996 #1
4997 \__seq_get_right_end:NnN \__seq_item:n
4998 }
4999 }
5000 \cs_new:Npn \__seq_get_right_loop:nw #1#2 \__seq_item:n
5001 {
5002 #2 \use_none:n {#1}
5003 \__seq_get_right_loop:nw
5004 }
5005 \cs_new:Npn \__seq_get_right_end:NnN #1#2#3 { \exp_not:n {#2} }
5006 \cs_generate_variant:Nn \seq_get_right:NN { c }

```

(End definition for `\seq_get_right:NN`, `__seq_get_right_loop:nw`, and `__seq_get_right_end:NnN`. This function is documented on page 67.)

`\seq_pop_right:NN` The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a “flexible length” way to set a token list variable. This is supplied by the `{ \if_false: } \fi: ... \if_false: { \fi: }` construct. Using an x-type expansion and a “non-expanding” definition for `__seq_item:n`, the left-most $n - 1$ entries in a sequence of n items are stored back in the sequence. That needs a loop of unknown length, hence using the strange `\if_false:` way of including braces. When the last item of the sequence is reached, the closing brace for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment. One more iteration is performed, with an empty argument and `\use_none:nn`, which finally stops the loop.

```

5007 \cs_new_protected:Npn \seq_pop_right:NN
5008 { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_set:Nx }
5009 \cs_new_protected:Npn \seq_gpop_right:NN
5010 { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_gset:Nx }

```

```

5011 \cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
5012 {
5013   \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
5014   \cs_set_eq:NN \__seq_item:n \scan_stop:
5015   #1 #2
5016   { \if_false: } \fi: \s__seq
5017     \exp_after:wN \use_i:nnn
5018     \exp_after:wN \__seq_pop_right_loop:nn
5019     #2
5020     {
5021       \if_false: { \fi: }
5022       \tl_set:Nx #3
5023     }
5024     { } \use_none:nn
5025     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
5026   }
5027 \cs_new:Npn \__seq_pop_right_loop:nn #1#2
5028 {
5029   #2 { \exp_not:n {#1} }
5030   \__seq_pop_right_loop:nn
5031 }
5032 \cs_generate_variant:Nn \seq_pop_right:NN { c }
5033 \cs_generate_variant:Nn \seq_gpop_right:NN { c }

```

(End definition for `\seq_pop_right:NN` and others. These functions are documented on page 67.)

`\seq_get_left:NNTF` Getting from the left or right with a check on the results. The first argument to `__seq_pop_TF:NNNN` is left unused.

```

\seq_get_left:cNTF
\seq_get_right:NNTF
\seq_get_right:cNTF
5034 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
5035 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
5036 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
5037 { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
5038 \prg_generate_conditional_variant:Nnn \seq_get_left:NN
5039 { c } { T , F , TF }
5040 \prg_generate_conditional_variant:Nnn \seq_get_right:NN
5041 { c } { T , F , TF }

```

(End definition for `\seq_get_left:NNTF` and `\seq_get_right:NNTF`. These functions are documented on page 68.)

`\seq_pop_left:NNTF` More or less the same for popping.

```

\seq_pop_left:cNTF
\seq_gpop_left:NNTF
\seq_gpop_left:cNTF
\seq_pop_right:NNTF
\seq_pop_right:cNTF
\seq_gpop_right:NNTF
\seq_gpop_right:cNTF
5042 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2
5043 { T , F , TF }
5044 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
5045 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2
5046 { T , F , TF }
5047 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
5048 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2
5049 { T , F , TF }
5050 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_set:Nx #1 #2 }
5051 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2
5052 { T , F , TF }
5053 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_gset:Nx #1 #2 }
5054 \prg_generate_conditional_variant:Nnn \seq_pop_left:NN { c }
5055 { T , F , TF }

```

```

5056 \prg_generate_conditional_variant:Nnn \seq_gpop_left:NN { c }
5057 { T , F , TF }
5058 \prg_generate_conditional_variant:Nnn \seq_gpop_right:NN { c }
5059 { T , F , TF }
5060 \prg_generate_conditional_variant:Nnn \seq_gpop_right:NN { c }
5061 { T , F , TF }

```

(End definition for `\seq_pop_left:NNTF` and others. These functions are documented on page 68.)

`\seq_item:Nn` The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then the argument delimited by `__seq_item:cn` is `\prg_break:` instead of being empty, terminating the loop and returning nothing at all.

```

\__seq_item:wNn
\__seq_item:nN
\__seq_item:nwn
5062 \cs_new:Npn \seq_item:Nn #1
5063 { \exp_after:wN \__seq_item:wNn #1 \q_stop #1 }
5064 \cs_new:Npn \__seq_item:wNn \s_seq #1 \q_stop #2#3
5065 {
5066   \exp_args:Nf \__seq_item:nwn
5067   { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
5068   #1
5069   \prg_break: \__seq_item:n { }
5070   \prg_break_point:
5071 }
5072 \cs_new:Npn \__seq_item:nN #1#2
5073 {
5074   \int_compare:nNnTF {#1} < 0
5075   { \int_eval:n { \seq_count:N #2 + 1 + #1 } }
5076   {#1}
5077 }
5078 \cs_new:Npn \__seq_item:nwn #1#2 \__seq_item:n #3
5079 {
5080   #2
5081   \int_compare:nNnTF {#1} = 1
5082   { \prg_break:n { \exp_not:n {#3} } }
5083   { \exp_args:Nf \__seq_item:nwn { \int_eval:n { #1 - 1 } } }
5084 }
5085 \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for `\seq_item:Nn` and others. This function is documented on page 67.)

9.6 Mapping to sequences

`\seq_map_break:` To break a function, the special token `\prg_break_point:Nn` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

```

5086 \cs_new:Npn \seq_map_break:
5087 { \prg_map_break:Nn \seq_map_break: { } }
5088 \cs_new:Npn \seq_map_break:n
5089 { \prg_map_break:Nn \seq_map_break: }

```

(End definition for `\seq_map_break:` and `\seq_map_break:n`. These functions are documented on page 70.)

`\seq_map_function:NN` The idea here is to apply the code of #2 to each item in the sequence without altering
`\seq_map_function:cN` the definition of `__seq_item:n`. The argument delimited by `__seq_item:n` is almost
`__seq_map_function:NNn` always empty, except at the end of the loop where it is `\prg_break:.` This allows to
break the loop without needing to do a (relatively-expensive) quark test.

```

5090 \cs_new:Npn \seq_map_function:NN #1#2
5091 {
5092   \exp_after:wN \use_i_ii:nnn
5093   \exp_after:wN \__seq_map_function:Nw
5094   \exp_after:wN #2
5095   #1
5096   \prg_break: \__seq_item:n { } \prg_break_point:
5097   \prg_break_point:Nn \seq_map_break: { }
5098 }
5099 \cs_new:Npn \__seq_map_function:Nw #1#2 \__seq_item:n #3
5100 {
5101   #2
5102   #1 {#3}
5103   \__seq_map_function:Nw #1
5104 }
5105 \cs_generate_variant:Nn \seq_map_function:NN { c }

```

(End definition for `\seq_map_function:NN` and `__seq_map_function:NNn`. This function is documented on page 70.)

`__seq_push_item_def:n` The definition of `__seq_item:n` needs to be saved and restored at various points within
`__seq_push_item_def:x` the mapping and manipulation code. That is handled here: as always, this approach uses
`__seq_push_item_def:` global assignments.
`__seq_pop_item_def:`

```

5106 \cs_new_protected:Npn \__seq_push_item_def:n
5107 {
5108   \__seq_push_item_def:
5109   \cs_gset:Npn \__seq_item:n ##1
5110 }
5111 \cs_new_protected:Npn \__seq_push_item_def:x
5112 {
5113   \__seq_push_item_def:
5114   \cs_gset:Npx \__seq_item:n ##1
5115 }
5116 \cs_new_protected:Npn \__seq_push_item_def:
5117 {
5118   \int_gincr:N \g__kernel_prg_map_int
5119   \cs_gset_eq:cN { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w }
5120   \__seq_item:n
5121 }
5122 \cs_new_protected:Npn \__seq_pop_item_def:
5123 {
5124   \cs_gset_eq:Nc \__seq_item:n
5125   { \__seq_map_ \int_use:N \g__kernel_prg_map_int :w }
5126   \int_gdecr:N \g__kernel_prg_map_int
5127 }

```

(End definition for `__seq_push_item_def:n`, `__seq_push_item_def:`, and `__seq_pop_item_def:.`)

`\seq_map_inline:NN` The idea here is that `__seq_item:n` is already “applied” to each item in a sequence,
`\seq_map_inline:cN` and so an in-line mapping is just a case of redefining `__seq_item:n`.

```

5128 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
5129 {
5130   \__seq_push_item_def:n {#2}
5131   #1
5132   \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
5133 }
5134 \cs_generate_variant:Nn \seq_map_inline:Nn { c }

```

(End definition for `\seq_map_inline:Nn`. This function is documented on page 70.)

`\seq_map_variable:NNn` This is just a specialised version of the in-line mapping function, using an x-type expansion for the code set up so that the number of # tokens required is as expected.

```

\seq_map_variable:Ncn
\seq_map_variable:cNn
\seq_map_variable:ccn
5135 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
5136 {
5137   \__seq_push_item_def:x
5138   {
5139     \tl_set:Nn \exp_not:N #2 {##1}
5140     \exp_not:n {#3}
5141   }
5142   #1
5143   \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
5144 }
5145 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
5146 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }

```

(End definition for `\seq_map_variable:NNn`. This function is documented on page 70.)

`\seq_count:N` Since counting the items in a sequence is quite common, we optimize it by grabbing 8 items at a time and correspondingly adding 8 to an integer expression. At the end of the loop, #9 is `__seq_count_end:w` instead of being empty. It removes 8+ and instead places the number of `__seq_item:n` that `__seq_count:w` grabbed before reaching the end of the sequence.

```

5147 \cs_new:Npn \seq_count:N #1
5148 {
5149   \int_eval:n
5150   {
5151     \exp_after:wN \use_i:nn
5152     \exp_after:wN \__seq_count:w
5153     #1
5154     \__seq_count_end:w \__seq_item:n 7
5155     \__seq_count_end:w \__seq_item:n 6
5156     \__seq_count_end:w \__seq_item:n 5
5157     \__seq_count_end:w \__seq_item:n 4
5158     \__seq_count_end:w \__seq_item:n 3
5159     \__seq_count_end:w \__seq_item:n 2
5160     \__seq_count_end:w \__seq_item:n 1
5161     \__seq_count_end:w \__seq_item:n 0
5162     \prg_break_point:
5163   }
5164 }
5165 \cs_new:Npn \__seq_count:w
5166   #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4 \__seq_item:n
5167   #5 \__seq_item:n #6 \__seq_item:n #7 \__seq_item:n #8 #9 \__seq_item:n
5168   { #9 8 + \__seq_count:w }

```

```

5169 \cs_new:Npn \__seq_count_end:w 8 + \__seq_count:w #1#2 \prg_break_point: {#1}
5170 \cs_generate_variant:Nn \seq_count:N { c }

```

(End definition for `\seq_count:N`, `__seq_count:w`, and `__seq_count_end:w`. This function is documented on page 71.)

9.7 Using sequences

```

\seq_use:Nnnn See \clist_use:Nnnn for a general explanation. The main difference is that we use \_
\seq_use:cnnn \__seq_item:n as a delimiter rather than commas. We also need to add \__seq_item:n
\__seq_use:NNnNnn at various places, and \s__seq.
\__seq_use_setup:w 5171 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
\__seq_use:nwwwnwn 5172 {
\__seq_use:nwwn 5173 \seq_if_exist:NTF #1
\seq_use:Nn 5174 {
\seq_use:cn 5175 \int_case:nnF { \seq_count:N #1 }
5176 {
5177 { 0 } { }
5178 { 1 } { \exp_after:wN \__seq_use:NNnNnn #1 ? { } { } }
5179 { 2 } { \exp_after:wN \__seq_use:NNnNnn #1 {#2} }
5180 }
5181 {
5182 \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
5183 \q_mark { \__seq_use:nwwwnwn {#3} }
5184 \q_mark { \__seq_use:nwwn {#4} }
5185 \q_stop { }
5186 }
5187 }
5188 {
5189 \__kernel_msg_expandable_error:nnn
5190 { kernel } { bad-variable } {#1}
5191 }
5192 }
5193 \cs_generate_variant:Nn \seq_use:Nnnn { c }
5194 \cs_new:Npn \__seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
5195 \cs_new:Npn \__seq_use_setup:w \s__seq { \__seq_use:nwwwnwn { } }
5196 \cs_new:Npn \__seq_use:nwwwnwn
5197 #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
5198 \q_mark #6#7 \q_stop #8
5199 {
5200 #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
5201 \q_mark {#6} #7 \q_stop { #8 #1 #2 }
5202 }
5203 \cs_new:Npn \__seq_use:nwwn #1 \__seq_item:n #2 #3 \q_stop #4
5204 { \exp_not:n { #4 #1 #2 } }
5205 \cs_new:Npn \seq_use:Nn #1#2
5206 { \seq_use:Nnnn #1 {#2} {#2} {#2} }
5207 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End definition for `\seq_use:Nnnn` and others. These functions are documented on page 71.)

9.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

\seq_push:Nn Pushing to a sequence is the same as adding on the left.

\seq_push:NV	5208	\cs_new_eq:NN	\seq_push:Nn	\seq_put_left:Nn
\seq_push:Nv	5209	\cs_new_eq:NN	\seq_push:NV	\seq_put_left:Nv
\seq_push:No	5210	\cs_new_eq:NN	\seq_push:Nv	\seq_put_left:Nv
\seq_push:Nx	5211	\cs_new_eq:NN	\seq_push:No	\seq_put_left:No
\seq_push:cn	5212	\cs_new_eq:NN	\seq_push:Nx	\seq_put_left:Nx
\seq_push:cV	5213	\cs_new_eq:NN	\seq_push:cn	\seq_put_left:cn
\seq_push:cV	5214	\cs_new_eq:NN	\seq_push:cV	\seq_put_left:cV
\seq_push:co	5215	\cs_new_eq:NN	\seq_push:cv	\seq_put_left:cv
\seq_push:cx	5216	\cs_new_eq:NN	\seq_push:co	\seq_put_left:co
\seq_gpush:Nn	5217	\cs_new_eq:NN	\seq_gpush:cx	\seq_put_left:cx
\seq_gpush:NV	5218	\cs_new_eq:NN	\seq_gpush:Nn	\seq_gput_left:Nn
\seq_gpush:Nv	5219	\cs_new_eq:NN	\seq_gpush:NV	\seq_gput_left:Nv
\seq_gpush:No	5220	\cs_new_eq:NN	\seq_gpush:Nv	\seq_gput_left:Nv
\seq_gpush:Nx	5221	\cs_new_eq:NN	\seq_gpush:No	\seq_gput_left:No
\seq_gpush:Nx	5222	\cs_new_eq:NN	\seq_gpush:Nx	\seq_gput_left:Nx
\seq_gpush:cn	5223	\cs_new_eq:NN	\seq_gpush:cn	\seq_gput_left:cn
\seq_gpush:cV	5224	\cs_new_eq:NN	\seq_gpush:cV	\seq_gput_left:cV
\seq_gpush:cv	5225	\cs_new_eq:NN	\seq_gpush:cv	\seq_gput_left:cv
\seq_gpush:co	5226	\cs_new_eq:NN	\seq_gpush:co	\seq_gput_left:co
\seq_gpush:cx	5227	\cs_new_eq:NN	\seq_gpush:cx	\seq_gput_left:cx

(End definition for **\seq_push:Nn** and **\seq_gpush:Nn**. These functions are documented on page 73.)

\seq_get:NN In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.

\seq_get:cN	
\seq_pop:NN	5228 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\seq_pop:cN	5229 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\seq_gpop:NN	5230 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
\seq_gpop:cN	5231 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
	5232 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
	5233 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN

(End definition for **\seq_get:NN**, **\seq_pop:NN**, and **\seq_gpop:NN**. These functions are documented on page 72.)

\seq_get:NNTF	More copies.
\seq_get:cNTF	5234 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
\seq_pop:NNTF	5235 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
\seq_pop:cNTF	5236 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
\seq_gpop:NNTF	5237 \prg_new_eq_conditional:NNn \seq_pop:cN \seq_pop_left:cN { T , F , TF }
\seq_gpop:cNTF	5238 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
	5239 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }

(End definition for **\seq_get:NNTF**, **\seq_pop:NNTF**, and **\seq_gpop:NNTF**. These functions are documented on page 72.)

9.9 Viewing sequences

\seq_show:N	Apply the general \msg_show:nnnnnn .
\seq_show:c	5240 \cs_new_protected:Npn \seq_show:N { _seq_show:NN \msg_show:nnxxxx }
\seq_log:N	5241 \cs_generate_variant:Nn \seq_show:N { c }
\seq_log:c	5242 \cs_new_protected:Npn \seq_log:N { _seq_show:NN \msg_log:nnxxxx }
_seq_show:NN	5243 \cs_generate_variant:Nn \seq_log:N { c }


```

5244 \cs_new_protected:Npn \__seq_show:NN #1#2
5245 {
5246   \__kernel_chk_defined:NT #2
5247   {
5248     #1 { LaTeX/kernel } { show-seq }
5249     { \token_to_str:N #2 }
5250     { \seq_map_function:NN #2 \msg_show_item:n }
5251     { } { }
5252   }
5253 }

```

(End definition for `\seq_show:N`, `\seq_log:N`, and `__seq_show:NN`. These functions are documented on page 75.)

9.10 Scratch sequences

`\l_tmpa_seq` Temporary comma list variables.

```

\l_tmpb_seq 5254 \seq_new:N \l_tmpa_seq
\g_tmpa_seq 5255 \seq_new:N \l_tmpb_seq
\g_tmpb_seq 5256 \seq_new:N \g_tmpa_seq
5257 \seq_new:N \g_tmpb_seq

```

(End definition for `\l_tmpa_seq` and others. These variables are documented on page 75.)

```
5258 </initex | package>
```

10 l3int implementation

```
5259 <*initex | package>
```

```
5260 <@@=int>
```

The following test files are used for this code: `m3int001`, `m3int002`, `m3int03`.

`\c_max_register_int` Done in l3basics.

(End definition for `\c_max_register_int`. This variable is documented on page 87.)

`__int_to_roman:w` Done in l3basics.

`\if_int_compare:w` (End definition for `__int_to_roman:w` and `\if_int_compare:w`. This function is documented on page 88.)

`\or:` Done in l3basics.

(End definition for `\or:`. This function is documented on page 89.)

`\int_value:w` Here are the remaining primitives for number comparisons and expressions.

```

\__int_eval:w 5261 \cs_new_eq:NN \int_value:w \tex_number:D
\__int_eval_end: 5262 \cs_new_eq:NN \__int_eval:w \etex_numexpr:D
\if_int_odd:w 5263 \cs_new_eq:NN \__int_eval_end: \tex_relax:D
\if_case:w 5264 \cs_new_eq:NN \if_int_odd:w \tex_ifodd:D
5265 \cs_new_eq:NN \if_case:w \tex_ifcase:D

```

(End definition for `\int_value:w` and others. These functions are documented on page 88.)

10.1 Integer expressions

`\int_eval:n` Wrapper for `__int_eval:w`: can be used in an integer expression or directly in the input stream. When debugging, use parentheses to catch early termination.

```

5266 \__kernel_patch_args:nNNpn
5267 { { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_eval:n } }
5268 \cs_new:Npn \int_eval:n #1
5269 { \int_value:w \__int_eval:w #1 \__int_eval_end: }
5270 \cs_new:Npn \int_eval:w { \int_value:w \__int_eval:w }
```

(End definition for `\int_eval:n` and `\int_eval:w`. These functions are documented on page 76.)

`\int_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

```

\__int_abs:N
\int_max:nn
\int_min:nn
\__int_maxmin:wwN
5271 \__kernel_patch_args:nNNpn
5272 { { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_abs:n } }
5273 \cs_new:Npn \int_abs:n #1
5274 {
5275   \int_value:w \exp_after:wN \__int_abs:N
5276   \int_value:w \__int_eval:w #1 \__int_eval_end:
5277   \exp_stop_f:
5278 }
5279 \cs_new:Npn \__int_abs:N #1
5280 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
5281 \__kernel_patch_args:nNNpn
5282 {
5283   { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_max:nn }
5284   { \__kernel_chk_expr:nNnN {#2} \__int_eval:w { } \int_max:nn }
5285 }
5286 \cs_set:Npn \int_max:nn #1#2
5287 {
5288   \int_value:w \exp_after:wN \__int_maxmin:wwN
5289   \int_value:w \__int_eval:w #1 \exp_after:wN ;
5290   \int_value:w \__int_eval:w #2 ;
5291   >
5292   \exp_stop_f:
5293 }
5294 \__kernel_patch_args:nNNpn
5295 {
5296   { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_min:nn }
5297   { \__kernel_chk_expr:nNnN {#2} \__int_eval:w { } \int_min:nn }
5298 }
5299 \cs_set:Npn \int_min:nn #1#2
5300 {
5301   \int_value:w \exp_after:wN \__int_maxmin:wwN
5302   \int_value:w \__int_eval:w #1 \exp_after:wN ;
5303   \int_value:w \__int_eval:w #2 ;
5304   <
5305   \exp_stop_f:
5306 }
5307 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
5308 {
5309   \if_int_compare:w #1 #3 #2 ~
5310   #1
```

```

5311     \else:
5312         #2
5313     \fi:
5314 }

```

(End definition for `\int_abs:n` and others. These functions are documented on page 76.)

`\int_div_truncate:nn` As `__int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expressions are expensive to evaluate (e.g., `\tl_count:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that 0/0 is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by $(| \#3\#4 | - 1)/2$, which we round away from zero. It turns out that this quantity exactly compensates the difference between ε -TeX's rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

```

5315 \__kernel_patch_args:nNpN
5316 {
5317     { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_div_truncate:nn }
5318     { \__kernel_chk_expr:nNnN {#2} \__int_eval:w { } \int_div_truncate:nn }
5319 }
5320 \cs_new:Npn \int_div_truncate:nn #1#2
5321 {
5322     \int_value:w \__int_eval:w
5323     \exp_after:wN \__int_div_truncate:NwNw
5324     \int_value:w \__int_eval:w #1 \exp_after:wN ;
5325     \int_value:w \__int_eval:w #2 ;
5326     \__int_eval_end:
5327 }
5328 \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
5329 {
5330     \if_meaning:w 0 #1
5331     0
5332     \else:
5333     (
5334         #1#2
5335         \if_meaning:w - #1 + \else: - \fi:
5336         ( \if_meaning:w - #3 - \fi: #3#4 - 1 ) / 2
5337     )
5338     \fi:
5339     / #3#4
5340 }

```

For the sake of completeness:

```

5341 \cs_new:Npn \int_div_round:nn #1#2
5342 { \int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }

```

Finally there's the modulus operation.

```

5343 \__kernel_patch_args:nNpN
5344 {
5345     { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_mod:nn }
5346     { \__kernel_chk_expr:nNnN {#2} \__int_eval:w { } \int_mod:nn }
5347 }
5348 \cs_new:Npn \int_mod:nn #1#2

```

```

5349 {
5350   \int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
5351   \int_value:w \__int_eval:w #1 \exp_after:wN ;
5352   \int_value:w \__int_eval:w #2 ;
5353   \__int_eval_end:
5354 }
5355 \cs_new:Npn \__int_mod:ww #1; #2;
5356 { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }

```

(End definition for `\int_div_truncate:nn` and others. These functions are documented on page 77.)

10.2 Creating and initialising integers

`\int_new:N` Two ways to do this: one for the format and one for the L^AT_EX 2_ε package. In plain T_EX, `\int_new:c` `\newcount` (and other allocators) are `\outer`: to allow the code here to work in “generic” mode this is therefore accessed by name. (The same applies to `\newbox`, `\newdimen` and so on.)

```

5357 \*package)
5358 \cs_new_protected:Npn \int_new:N #1
5359 {
5360   \__kernel_chk_if_free_cs:N #1
5361   \cs:w newcount \cs_end: #1
5362 }
5363 \*package)
5364 \cs_generate_variant:Nn \int_new:N { c }

```

(End definition for `\int_new:N`. This function is documented on page 77.)

`\int_const:Nn` As stated, most constants can be defined as `\chardef` or `\mathchardef` but that’s engine dependent. As a result, there is some set up code to determine what can be done. No full engine testing just yet so everything is a little awkward. We cannot use `\int_gset:Nn` because (when `check-declarations` is enabled) this runs some checks that constants would fail.

```

5365 \__kernel_patch_args:nnnNnNpn
5366 { \__kernel_chk_var_scope:NN c #1 }
5367 { }
5368 { {#1} { \__kernel_chk_expr:nNnN {#2} \__int_eval:w { } \int_const:Nn } }
5369 \cs_new_protected:Npn \int_const:Nn #1#2
5370 {
5371   \int_compare:nNnTF {#2} < \c_zero
5372   {
5373     \int_new:N #1
5374     \tex_global:D
5375   }
5376   {
5377     \int_compare:nNnTF {#2} > \c__int_max_constdef_int
5378     {
5379       \int_new:N #1
5380       \tex_global:D
5381     }
5382     {
5383       \__kernel_chk_if_free_cs:N #1
5384       \tex_global:D \__int_constdef:Nw
5385     }
5386   }

```

```

5386     }
5387     #1 = \__int_eval:w #2 \__int_eval_end:
5388   }
5389   \cs_generate_variant:Nn \int_const:Nn { c }
5390   \if_int_odd:w 0
5391     \cs_if_exist:NT \luatex luatexversion:D { 1 }
5392     \cs_if_exist:NT \uptex_disablecjktoken:D
5393       { \if_int_compare:w \ptex_jis:D "2121 = "3000 ~ 1 \fi: }
5394     \cs_if_exist:NT \xetex_XeTeXversion:D { 1 } ~
5395     \cs_if_exist:NTF \uptex_disablecjktoken:D
5396       { \cs_new_eq:NN \__int_constdef:Nw \uptex_kchardef:D }
5397       { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
5398     \__int_constdef:Nw \c__int_max_constdef_int 1114111 ~
5399   \else:
5400     \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
5401     \tex_mathchardef:D \c__int_max_constdef_int 32767 ~
5402   \fi:

```

(End definition for `\int_const:Nn`, `__int_constdef:Nw`, and `\c__int_max_constdef_int`. This function is documented on page 77.)

```

\int_zero:N Functions that reset an <integer> register to zero.
\int_zero:c 5403 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\int_gzero:N 5404 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero }
\int_gzero:c 5405 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
5406 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero }
5407 \cs_generate_variant:Nn \int_zero:N { c }
5408 \cs_generate_variant:Nn \int_gzero:N { c }

```

(End definition for `\int_zero:N` and `\int_gzero:N`. These functions are documented on page 77.)

```

\int_zero_new:N Create a register if needed, otherwise clear it.
\int_zero_new:c 5409 \cs_new_protected:Npn \int_zero_new:N #1
\int_gzero_new:N 5410 { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
\int_gzero_new:c 5411 \cs_new_protected:Npn \int_gzero_new:N #1
5412 { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
5413 \cs_generate_variant:Nn \int_zero_new:N { c }
5414 \cs_generate_variant:Nn \int_gzero_new:N { c }

```

(End definition for `\int_zero_new:N` and `\int_gzero_new:N`. These functions are documented on page 77.)

```

\int_set_eq:NN Setting equal means using one integer inside the set function of another. Check that
\int_set_eq:cN assigned integer is local/global. No need to check that the other one is defined as TeX
\int_set_eq:Nc does it for us.
\int_set_eq:cc 5415 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\int_gset_eq:NN 5416 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
\int_gset_eq:cN 5417 \cs_generate_variant:Nn \int_set_eq:NN { c , Nc , cc }
\int_gset_eq:Nc 5418 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
\int_gset_eq:cc 5419 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
5420 \cs_generate_variant:Nn \int_gset_eq:NN { c , Nc , cc }

```

(End definition for `\int_set_eq:NN` and `\int_gset_eq:NN`. These functions are documented on page 78.)

```

\int_if_exist_p:N Copies of the cs functions defined in l3basics.
\int_if_exist_p:c 5421 \prg_new_eq_conditional:Nn \int_if_exist:N \cs_if_exist:N
\int_if_exist:NTF 5422 { TF , T , F , p }
\int_if_exist:cTF 5423 \prg_new_eq_conditional:Nn \int_if_exist:c \cs_if_exist:c
5424 { TF , T , F , p }

```

(End definition for `\int_if_exist:N`. This function is documented on page 78.)

10.3 Setting and incrementing integers

Several functions here have a signature `:Nn` and are such that when debugging, the first argument should be checked to be a local/global variable and the second should be wrapped in code for an expression. The temporary function `__int_tmp:w` finds the name `#3` of the function being redefined and writes the appropriate patch.

```

5425 \cs_set_protected:Npn \__int_tmp:w #1#2#3
5426 {
5427   \__kernel_patch_args:nnnNnNpn
5428   { #1 ##1 }
5429   { }
5430   { {##1} { \__kernel_chk_expr:nNnN {##2} \__int_eval:w { } #3 } }
5431   #2 #3
5432 }

```

```

\int_add:Nn Adding and subtracting to and from a counter. For each function, the debugging code
\int_add:cn produced by \__int_tmp:w checks that the assigned variable is correctly local/global and
\int_gadd:Nn wraps the expression in some checking code.
\int_gadd:cn 5433 \__int_tmp:w \__kernel_chk_var_local:N
\int_sub:Nn 5434 \cs_new_protected:Npn \int_add:Nn #1#2
\int_sub:cn 5435 { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
\int_gsub:Nn 5436 \__int_tmp:w \__kernel_chk_var_local:N
\int_gsub:cn 5437 \cs_new_protected:Npn \int_sub:Nn #1#2
5438 { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
5439 \__int_tmp:w \__kernel_chk_var_global:N
5440 \cs_new_protected:Npn \int_gadd:Nn #1#2
5441 { \tex_global:D \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
5442 \__int_tmp:w \__kernel_chk_var_global:N
5443 \cs_new_protected:Npn \int_gsub:Nn #1#2
5444 { \tex_global:D \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
5445 \cs_generate_variant:Nn \int_add:Nn { c }
5446 \cs_generate_variant:Nn \int_gadd:Nn { c }
5447 \cs_generate_variant:Nn \int_sub:Nn { c }
5448 \cs_generate_variant:Nn \int_gsub:Nn { c }

```

(End definition for `\int_add:Nn` and others. These functions are documented on page 78.)

```

\int_incr:N Incrementing and decrementing of integer registers is done with the following functions.
\int_incr:c 5449 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\int_gincr:N 5450 \cs_new_protected:Npn \int_incr:N #1
\int_gincr:c 5451 { \tex_advance:D #1 \c_one }
\int_decr:N 5452 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\int_decr:c 5453 \cs_new_protected:Npn \int_decr:N #1
\int_gdecr:N 5454 { \tex_advance:D #1 - \c_one }
\int_gdecr:c 5455 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }

```

```

5456 \cs_new_protected:Npn \int_gincr:N #1
5457 { \tex_global:D \tex_advance:D #1 \c_one }
5458 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
5459 \cs_new_protected:Npn \int_gdecr:N #1
5460 { \tex_global:D \tex_advance:D #1 - \c_one }
5461 \cs_generate_variant:Nn \int_incr:N { c }
5462 \cs_generate_variant:Nn \int_decr:N { c }
5463 \cs_generate_variant:Nn \int_gincr:N { c }
5464 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End definition for `\int_incr:N` and others. These functions are documented on page 78.)

`\int_set:Nn` As integers are register-based TeX issues an error if they are not defined. Thus there is no need to check their existence as for token list variables. However, the code that checks whether the assignment is local or global is still needed.

```

\int_set:cn
\int_gset:Nn
\int_gset:cn
5465 \__int_tmp:w \__kernel_chk_var_local:N
5466 \cs_new_protected:Npn \int_set:Nn #1#2
5467 { #1 ~ \__int_eval:w #2 \__int_eval_end: }
5468 \__int_tmp:w \__kernel_chk_var_global:N
5469 \cs_new_protected:Npn \int_gset:Nn #1#2
5470 { \tex_global:D #1 ~ \__int_eval:w #2 \__int_eval_end: }
5471 \cs_generate_variant:Nn \int_set:Nn { c }
5472 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for `\int_set:Nn` and `\int_gset:Nn`. These functions are documented on page 78.)

10.4 Using integers

`\int_use:N` Here is how counters are accessed:

```

\int_use:c
5473 \cs_new_eq:NN \int_use:N \tex_the:D

```

We hand-code this for some speed gain:

```

5474 %\cs_generate_variant:Nn \int_use:N { c }
5475 \cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End definition for `\int_use:N`. This function is documented on page 79.)

10.5 Integer expression conditionals

`__int_compare_error:` Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. The tests first evaluate their left-hand side, with a trailing `__int_compare_error:.` This marker is normally not expanded, but if the relation symbol is missing from the test's argument, then the marker inserts `=` (and itself) after triggering the relevant TeX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed `__int_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```

5476 \cs_new_protected:Npn \__int_compare_error:
5477 {
5478   \if_int_compare:w \c_zero \c_zero \fi:
5479   =
5480   \__int_compare_error:
5481 }
5482 \cs_new:Npn \__int_compare_error:Nw

```

```

5483     #1#2 \q_stop
5484     {
5485     { }
5486     \c_zero \fi:
5487     \__kernel_msg_expandable_error:nnn
5488     { kernel } { unknown-comparison } {#1}
5489     \prg_return_false:
5490     }

```

(End definition for __int_compare_error: and __int_compare_error:Nw.)

`\int_compare_p:n` Comparison tests using a simple syntax where only one set of braces is required, additional operators such as `!=` and `>=` are supported, and multiple comparisons can be performed at once, for instance `0 < 5 <= 1`. The idea is to loop through the argument, finding one operand at a time, and comparing it to the previous one. The looping auxiliary `__int_compare:Nw` reads one *<operand>* and one *<comparison>* symbol, and leaves roughly

```

\__int_compare:w      <operand> \prg_return_false: \fi:
\__int_compare:Nw     \reverse_if:N \if_int_compare:w <operand> <comparison>
\__int_compare:NNw    \__int_compare:Nw
\__int_compare:nnN
\__int_compare_end=:NNw
\__int_compare_=:NNw
\__int_compare_<:NNw
\__int_compare_>:NNw
\__int_compare_=:NNw
\__int_compare_!=:NNw
\__int_compare_<=:NNw
\__int_compare_>=:NNw

```

in the input stream. Each call to this auxiliary provides the second operand of the last call's `\if_int_compare:w`. If one of the *<comparisons>* is `false`, the `true` branch of the T_EX conditional is taken (because of `\reverse_if:N`), immediately returning `false` as the result of the test. There is no T_EX conditional waiting the first operand, so we add an `\if_false:` and expand by hand with `\int_value:w`, thus skipping `\prg_return_false:` on the first iteration.

Before starting the loop, the first step is to make sure that there is at least one relation symbol. We first let T_EX evaluate this left hand side of the (in)equality using `__int_eval:w`. Since the relation symbols `<`, `>`, `=` and `!` are not allowed in integer expressions, they would terminate the expression. If the argument contains no relation symbol, `__int_compare_error:` is expanded, inserting `=` and itself after an error. In all cases, `__int_compare:w` receives as its argument an integer, a relation symbol, and some more tokens. We then setup the loop, which is ended by the two odd-looking items `e` and `{=nd_}`, with a trailing `\q_stop` used to grab the entire argument when necessary.

```

5491 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
5492 {
5493   \exp_after:wN \__int_compare:w
5494   \int_value:w \__int_eval:w #1 \__int_compare_error:
5495 }
5496 \cs_new:Npn \__int_compare:w #1 \__int_compare_error:
5497 {
5498   \exp_after:wN \if_false: \int_value:w
5499   \__int_compare:Nw #1 e { = nd_ } \q_stop
5500 }

```

The goal here is to find an *<operand>* and a *<comparison>*. The *<operand>* is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying `__int_to_roman:w`, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with `__int_compare:NNw` to determine the relation symbol, building a control sequence from it (`\token_to_str:N` gives better errors if `#1` is not a character). All the extended forms have an extra `=` hence the test for that as a

second token. If the relation symbol is unknown, then the control sequence is turned by \TeX into `\scan_stop:`, ignored thanks to `\unexpanded`, and `__int_compare_error:Nw` raises an error.

```

5501 \cs_new:Npn \__int_compare:Nw #1#2 \q_stop
5502 {
5503   \exp_after:wN \__int_compare:NNw
5504   \__int_to_roman:w - 0 #2 \q_mark
5505   #1#2 \q_stop
5506 }
5507 \cs_new:Npn \__int_compare:NNw #1#2#3 \q_mark
5508 {
5509   \__kernel_exp_not:w
5510   \use:c
5511   {
5512     \__int_compare_ \token_to_str:N #1
5513     \if_meaning:w = #2 = \fi:
5514     :NNw
5515   }
5516   \__int_compare_error:Nw #1
5517 }

```

When the last *operand* is seen, `__int_compare:NNw` receives `e` and `=nd_` as arguments, hence calling `__int_compare_end_=:NNw` to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls `__int_compare:nnN` where `#1` is `\if_int_compare:w` or `\reverse_if:N \if_int_compare:w`, `#2` is the *operand*, and `#3` is one of `<`, `=`, or `>`. As announced earlier, we leave the *operand* for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return `false`. Otherwise, we apply the conditional `#1` to the *operand* `#2` and the comparison `#3`, and call `__int_compare:Nw` to look for additional operands, after evaluating the following expression.

```

5518 \cs_new:cpn { \__int_compare_end_=:NNw } #1#2#3 e #4 \q_stop
5519 {
5520   {#3} \exp_stop_f:
5521   \prg_return_false: \else: \prg_return_true: \fi:
5522 }
5523 \cs_new:Npn \__int_compare:nnN #1#2#3
5524 {
5525   {#2} \exp_stop_f:
5526   \prg_return_false: \exp_after:wN \use_none_delimit_by_q_stop:w
5527   \fi:
5528   #1 #2 #3 \exp_after:wN \__int_compare:Nw \int_value:w \__int_eval:w
5529 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `__int_compare_error:Nw` *token* responsible for error detection.

```

5530 \cs_new:cpn { \__int_compare_=:NNw } #1#2#3 =
5531 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
5532 \cs_new:cpn { \__int_compare_<:NNw } #1#2#3 <
5533 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
5534 \cs_new:cpn { \__int_compare_>:NNw } #1#2#3 >
5535 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }

```

```

5536 \cs_new:cpn { __int_compare_==:NNw } #1#2#3 ==
5537 { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
5538 \cs_new:cpn { __int_compare_!=:NNw } #1#2#3 !=
5539 { \__int_compare:nnN { \if_int_compare:w } {#3} = }
5540 \cs_new:cpn { __int_compare_<=:NNw } #1#2#3 <=
5541 { \__int_compare:nnN { \if_int_compare:w } {#3} > }
5542 \cs_new:cpn { __int_compare_>=:NNw } #1#2#3 >=
5543 { \__int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End definition for \int_compare:nTF and others. This function is documented on page 80.)

\int_compare_p:nNn More efficient but less natural in typing.

```

\int_compare:nNnTF
5544 \__kernel_patch_conditional_args:nNnpnn
5545 {
5546   { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_compare:nNn }
5547   { \__int_eval_end: #2 }
5548   { \__kernel_chk_expr:nNnN {#3} \__int_eval:w { } \int_compare:nNn }
5549 }
5550 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
5551 {
5552   \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
5553   \prg_return_true:
5554   \else:
5555     \prg_return_false:
5556   \fi:
5557 }

```

(End definition for \int_compare:nNnTF. This function is documented on page 79.)

\int_case:nn For integer cases, the first task to fully expand the check condition. The over all idea is then much the same as for \tl_case:nn(TF) as described in l3tl.

```

\int_case:nnTF
\__int_case:nw
\__int_case_end:nw
5558 \cs_new:Npn \int_case:nnTF #1
5559 {
5560   \exp:w
5561   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
5562 }
5563 \cs_new:Npn \int_case:nnT #1#2#3
5564 {
5565   \exp:w
5566   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
5567 }
5568 \cs_new:Npn \int_case:nnF #1#2
5569 {
5570   \exp:w
5571   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
5572 }
5573 \cs_new:Npn \int_case:nn #1#2
5574 {
5575   \exp:w
5576   \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
5577 }
5578 \cs_new:Npn \__int_case:nnTF #1#2#3#4
5579 { \__int_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
5580 \cs_new:Npn \__int_case:nw #1#2#3

```

```

5581 {
5582   \int_compare:nNnTF {#1} = {#2}
5583     { \__int_case_end:nw {#3} }
5584     { \__int_case:nw {#1} }
5585 }
5586 \cs_new:Npn \__int_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
5587 { \exp_end: #1 #4 }

```

(End definition for `\int_case:nnTF` and others. This function is documented on page 81.)

```

\int_if_odd_p:n A predicate function.
\int_if_odd:nTF 5588 \__kernel_patch_conditional_args:nNnpnn
\int_if_even_p:n 5589 { { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_if_odd:n } }
\int_if_even:nTF 5590 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
5591 {
5592   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
5593     \prg_return_true:
5594   \else:
5595     \prg_return_false:
5596   \fi:
5597 }
5598 \__kernel_patch_conditional_args:nNnpnn
5599 { { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_if_even:n } }
5600 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
5601 {
5602   \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
5603     \prg_return_false:
5604   \else:
5605     \prg_return_true:
5606   \fi:
5607 }

```

(End definition for `\int_if_odd:nTF` and `\int_if_even:nTF`. These functions are documented on page 81.)

10.6 Integer expression loops

`\int_while_do:nn` These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

```

\int_until_do:nn 5608 \cs_new:Npn \int_while_do:nn #1#2
\int_do_while:nn 5609 {
\int_do_until:nn 5610   \int_compare:nT {#1}
5611     {
5612       #2
5613       \int_while_do:nn {#1} {#2}
5614     }
5615   }
5616 \cs_new:Npn \int_until_do:nn #1#2
5617 {
5618   \int_compare:nF {#1}
5619     {
5620       #2
5621       \int_until_do:nn {#1} {#2}
5622     }

```

```

5623     }
5624 \cs_new:Npn \int_do_while:nn #1#2
5625 {
5626     #2
5627     \int_compare:nT {#1}
5628     { \int_do_while:nn {#1} {#2} }
5629 }
5630 \cs_new:Npn \int_do_until:nn #1#2
5631 {
5632     #2
5633     \int_compare:nF {#1}
5634     { \int_do_until:nn {#1} {#2} }
5635 }

```

(End definition for `\int_while_do:nn` and others. These functions are documented on page 82.)

```

\int_while_do:nNnn As above but not using the more natural syntax.
\int_until_do:nNnn
\int_do_while:nNnn
\int_do_until:nNnn
5636 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
5637 {
5638     \int_compare:nNnT {#1} #2 {#3}
5639     {
5640         #4
5641         \int_while_do:nNnn {#1} #2 {#3} {#4}
5642     }
5643 }
5644 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
5645 {
5646     \int_compare:nNnF {#1} #2 {#3}
5647     {
5648         #4
5649         \int_until_do:nNnn {#1} #2 {#3} {#4}
5650     }
5651 }
5652 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
5653 {
5654     #4
5655     \int_compare:nNnT {#1} #2 {#3}
5656     { \int_do_while:nNnn {#1} #2 {#3} {#4} }
5657 }
5658 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
5659 {
5660     #4
5661     \int_compare:nNnF {#1} #2 {#3}
5662     { \int_do_until:nNnn {#1} #2 {#3} {#4} }
5663 }

```

(End definition for `\int_while_do:nNnn` and others. These functions are documented on page 82.)

10.7 Integer step functions

```

\int_step_function:nnnN Before all else, evaluate the initial value, step, and final value. Repeating a function by
    \__int_step:wwwN steps first needs a check on the direction of the steps. After that, do the function for the
    \__int_step:NwnnN start value then step and loop around. It would be more symmetrical to test for a step
\int_step_function:nN
\int_step_function:nnN

```

size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

5664 \__kernel_patch_args:nNnNpn
5665 {
5666     {
5667         \__kernel_chk_expr:nNnN {#1} \__int_eval:w { }
5668         \int_step_function:nnnN
5669     }
5670     {
5671         \__kernel_chk_expr:nNnN {#2} \__int_eval:w { }
5672         \int_step_function:nnnN
5673     }
5674     {
5675         \__kernel_chk_expr:nNnN {#3} \__int_eval:w { }
5676         \int_step_function:nnnN
5677     }
5678 }
5679 \cs_new:Npn \int_step_function:nnnN #1#2#3
5680 {
5681     \exp_after:wN \__int_step:wwwN
5682     \int_value:w \__int_eval:w #1 \exp_after:wN ;
5683     \int_value:w \__int_eval:w #2 \exp_after:wN ;
5684     \int_value:w \__int_eval:w #3 ;
5685 }
5686 \cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
5687 {
5688     \int_compare:nNnTF {#2} > \c_zero
5689     { \__int_step:NwnnN > }
5690     {
5691         \int_compare:nNnTF {#2} = \c_zero
5692         {
5693             \__kernel_msg_expandable_error:nnn
5694             { kernel } { zero-step } {#4}
5695             \prg_break:
5696         }
5697         { \__int_step:NwnnN < }
5698     }
5699     #1 ; {#2} {#3} #4
5700     \prg_break_point:
5701 }
5702 \cs_new:Npn \__int_step:NwnnN #1#2 ; #3#4#5
5703 {
5704     \if_int_compare:w #2 #1 #4 \exp_stop_f:
5705     \prg_break:n
5706     \fi:
5707     #5 {#2}
5708     \exp_after:wN \__int_step:NwnnN
5709     \exp_after:wN #1
5710     \int_value:w \__int_eval:w #2 + #3 ; {#3} {#4} #5
5711 }
5712 \cs_new:Npn \int_step_function:nN
5713 { \int_step_function:nnnN { 1 } { 1 } }
5714 \cs_new:Npn \int_step_function:nnN #1
5715 { \int_step_function:nnnN {#1} { 1 } }

```

(End definition for `\int_step_function:nnnN` and others. These functions are documented on page 83.)

`\int_step_inline:nn` The approach here is to build a function, with a global integer required to make the
`\int_step_inline:nnn` nesting safe (as seen in other in line functions), and map that function using `\int_-`
`\int_step_inline:nnnn` `step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions
`\int_step_variable:nNn` from other modules correctly decrement `\g__kernel_prg_map_int` before looking for
`\int_step_variable:nnNn` their own break point. The first argument is `\scan_stop:`, so that no breaking function
`\int_step_variable:nnnNn` recognizes this break point as its own.

```

5716 \cs_new_protected:Npn \int_step_inline:nn
5717   { \int_step_inline:nnnn { 1 } { 1 } }
5718 \cs_new_protected:Npn \int_step_inline:nnn #1
5719   { \int_step_inline:nnnn {#1} { 1 } }
5720 \cs_new_protected:Npn \int_step_inline:nnnn
5721   {
5722     \int_gincr:N \g__kernel_prg_map_int
5723     \exp_args:NNc \__int_step:NNnnnn
5724     \cs_gset_protected:Npn
5725       { __int_map_ \int_use:N \g__kernel_prg_map_int :w }
5726   }
5727 \cs_new_protected:Npn \int_step_variable:nNn
5728   { \int_step_variable:nnnNn { 1 } { 1 } }
5729 \cs_new_protected:Npn \int_step_variable:nnNn #1
5730   { \int_step_variable:nnnNn {#1} { 1 } }
5731 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
5732   {
5733     \int_gincr:N \g__kernel_prg_map_int
5734     \exp_args:NNc \__int_step:NNnnnn
5735     \cs_gset_protected:Npx
5736       { __int_map_ \int_use:N \g__kernel_prg_map_int :w }
5737       {#1}{#2}{#3}
5738       {
5739         \tl_set:Nn \exp_not:N #4 {##1}
5740         \exp_not:n {#5}
5741       }
5742   }
5743 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
5744   {
5745     #1 #2 ##1 {#6}
5746     \int_step_function:nnnN {#3} {#4} {#5} #2
5747     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
5748   }

```

(End definition for `\int_step_inline:nn` and others. These functions are documented on page 83.)

10.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```

5749 \cs_new_eq:NN \int_to_arabic:n \int_eval:n

```

(End definition for `\int_to_arabic:n`. This function is documented on page 84.)

`\int_to_symbols:nnn` For conversion of integers to arbitrary symbols the method is in general as follows. The
`__int_to_symbols:nnnn` input number (#1) is compared to the total number of symbols available at each place
 (#2). If the input is larger than the total number of symbols available then the modulus

is needed, with one added so that the positions don't have to number from zero. Using an **f**-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```

5750 \cs_new:Npn \int_to_symbols:nnn #1#2#3
5751 {
5752   \int_compare:nNnTF {#1} > {#2}
5753   {
5754     \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
5755     {
5756       \int_case:nn
5757       { 1 + \int_mod:nn { #1 - 1 } {#2} }
5758       {#3}
5759     }
5760     {#1} {#2} {#3}
5761   }
5762   { \int_case:nn {#1} {#3} }
5763 }
5764 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
5765 {
5766   \exp_args:Nf \int_to_symbols:nnn
5767   { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
5768   #1
5769 }

```

(End definition for `\int_to_symbols:nnn` and `__int_to_symbols:nnnn`. This function is documented on page 84.)

`\int_to_alph:n` These both use the above function with input functions that make sense for the alphabet
`\int_to_Alph:n` in English.

```

5770 \cs_new:Npn \int_to_alph:n #1
5771 {
5772   \int_to_symbols:nnn {#1} { 26 }
5773   {
5774     { 1 } { a }
5775     { 2 } { b }
5776     { 3 } { c }
5777     { 4 } { d }
5778     { 5 } { e }
5779     { 6 } { f }
5780     { 7 } { g }
5781     { 8 } { h }
5782     { 9 } { i }
5783     { 10 } { j }
5784     { 11 } { k }
5785     { 12 } { l }
5786     { 13 } { m }
5787     { 14 } { n }
5788     { 15 } { o }
5789     { 16 } { p }
5790     { 17 } { q }
5791     { 18 } { r }
5792     { 19 } { s }
5793     { 20 } { t }

```

```

5794         { 21 } { u }
5795         { 22 } { v }
5796         { 23 } { w }
5797         { 24 } { x }
5798         { 25 } { y }
5799         { 26 } { z }
5800     }
5801 }
5802 \cs_new:Npn \int_to_Alph:n #1
5803 {
5804     \int_to_symbols:nnn {#1} { 26 }
5805     {
5806         { 1 } { A }
5807         { 2 } { B }
5808         { 3 } { C }
5809         { 4 } { D }
5810         { 5 } { E }
5811         { 6 } { F }
5812         { 7 } { G }
5813         { 8 } { H }
5814         { 9 } { I }
5815         { 10 } { J }
5816         { 11 } { K }
5817         { 12 } { L }
5818         { 13 } { M }
5819         { 14 } { N }
5820         { 15 } { O }
5821         { 16 } { P }
5822         { 17 } { Q }
5823         { 18 } { R }
5824         { 19 } { S }
5825         { 20 } { T }
5826         { 21 } { U }
5827         { 22 } { V }
5828         { 23 } { W }
5829         { 24 } { X }
5830         { 25 } { Y }
5831         { 26 } { Z }
5832     }
5833 }

```

(End definition for `\int_to_alph:n` and `\int_to_Alph:n`. These functions are documented on page 84.)

`\int_to_base:nn` Converting from base ten (`#1`) to a second base (`#2`) starts with computing `#1`: if it is a complicated calculation, we shouldn't perform it twice. Then check the sign, store it, either `-` or `\c_empty_tl`, and feed the absolute value to the next auxiliary function.

`\int_to_Base:nn`

```

5834 \cs_new:Npn \int_to_base:nn #1
5835 { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
5836 \cs_new:Npn \int_to_Base:nn #1
5837 { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
5838 \cs_new:Npn \__int_to_base:nn #1#2
5839 {
5840     \int_compare:nNnTF {#1} < 0
5841     { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }

```



```

5842     { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
5843   }
5844 \cs_new:Npn \__int_to_Base:nn #1#2
5845 {
5846   \int_compare:nNnTF {#1} < 0
5847     { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
5848     { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
5849 }

```

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in #1 is checked to see if it is less than the new base (#2). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

```

5850 \cs_new:Npn \__int_to_base:nnN #1#2#3
5851 {
5852   \int_compare:nNnTF {#1} < {#2}
5853     { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
5854     {
5855       \exp_args:Nf \__int_to_base:nnnN
5856         { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
5857         {#1}
5858         {#2}
5859         #3
5860     }
5861 }
5862 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4
5863 {
5864   \exp_args:Nf \__int_to_base:nnN
5865     { \int_div_truncate:nn {#2} {#3} }
5866     {#3}
5867   #4
5868   #1
5869 }
5870 \cs_new:Npn \__int_to_Base:nnN #1#2#3
5871 {
5872   \int_compare:nNnTF {#1} < {#2}
5873     { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
5874     {
5875       \exp_args:Nf \__int_to_Base:nnnN
5876         { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
5877         {#1}
5878         {#2}
5879         #3
5880     }
5881 }
5882 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
5883 {
5884   \exp_args:Nf \__int_to_Base:nnN
5885     { \int_div_truncate:nn {#2} {#3} }
5886     {#3}
5887   #4
5888   #1

```

```
5889 }
```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\int_case:nn`, but in our case, the cases are contiguous, so it is forty times faster to use the `\if_case:w` primitive. The first `\exp_after:wN` expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since `#1` might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing `\fi:`.

```
5890 \cs_new:Npn \__int_to_letter:n #1
5891 {
5892   \exp_after:wN \exp_after:wN
5893   \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
5894     a
5895   \or: b
5896   \or: c
5897   \or: d
5898   \or: e
5899   \or: f
5900   \or: g
5901   \or: h
5902   \or: i
5903   \or: j
5904   \or: k
5905   \or: l
5906   \or: m
5907   \or: n
5908   \or: o
5909   \or: p
5910   \or: q
5911   \or: r
5912   \or: s
5913   \or: t
5914   \or: u
5915   \or: v
5916   \or: w
5917   \or: x
5918   \or: y
5919   \or: z
5920   \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
5921   \fi:
5922 }
5923 \cs_new:Npn \__int_to_Letter:n #1
5924 {
5925   \exp_after:wN \exp_after:wN
5926   \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
5927     A
5928   \or: B
5929   \or: C
5930   \or: D
5931   \or: E
5932   \or: F
5933   \or: G
5934   \or: H
5935   \or: I
```

```

5936     \or: J
5937     \or: K
5938     \or: L
5939     \or: M
5940     \or: N
5941     \or: O
5942     \or: P
5943     \or: Q
5944     \or: R
5945     \or: S
5946     \or: T
5947     \or: U
5948     \or: V
5949     \or: W
5950     \or: X
5951     \or: Y
5952     \or: Z
5953     \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
5954     \fi:
5955 }

```

(End definition for `\int_to_base:nn` and others. These functions are documented on page 85.)

`\int_to_bin:n` Wrappers around the generic function.

```

\int_to_hex:n 5956 \cs_new:Npn \int_to_bin:n #1
\int_to_Hex:n 5957 { \int_to_base:nn {#1} { 2 } }
\int_to_oct:n 5958 \cs_new:Npn \int_to_hex:n #1
5959 { \int_to_base:nn {#1} { 16 } }
5960 \cs_new:Npn \int_to_Hex:n #1
5961 { \int_to_Base:nn {#1} { 16 } }
5962 \cs_new:Npn \int_to_oct:n #1
5963 { \int_to_base:nn {#1} { 8 } }

```

(End definition for `\int_to_bin:n` and others. These functions are documented on page 85.)

`\int_to_roman:n` The `__int_to_roman:w` primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop is terminated by the conversion of the Q.

```

\__int_to_roman:N 5964 \cs_new:Npn \int_to_roman:n #1
\__int_to_roman:N 5965 {
\__int_to_roman_i:w 5966   \exp_after:wN \__int_to_roman:N
\__int_to_roman_v:w 5967   \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_roman_x:w 5968 }
\__int_to_roman_l:w 5969 \cs_new:Npn \__int_to_roman:N #1
\__int_to_roman_c:w 5970 {
\__int_to_roman_d:w 5971   \use:c { __int_to_roman_ #1 :w }
\__int_to_roman_m:w 5972   \__int_to_roman:N
\__int_to_roman_Q:w 5973 }
\__int_to_Roman_i:w 5974 \cs_new:Npn \int_to_Roman:n #1
\__int_to_Roman_v:w 5975 {
\__int_to_Roman_x:w 5976   \exp_after:wN \__int_to_Roman_aux:N
\__int_to_Roman_l:w 5977   \__int_to_roman:w \int_eval:n {#1} Q
\__int_to_Roman_c:w 5978 }
\__int_to_Roman_d:w
\__int_to_Roman_m:w
\__int_to_Roman_Q:w

```

```

5979 \cs_new:Npn \__int_to_Roman_aux:N #1
5980 {
5981   \use:c { __int_to_Roman_ #1 :w }
5982   \__int_to_Roman_aux:N
5983 }
5984 \cs_new:Npn \__int_to_roman_i:w { i }
5985 \cs_new:Npn \__int_to_roman_v:w { v }
5986 \cs_new:Npn \__int_to_roman_x:w { x }
5987 \cs_new:Npn \__int_to_roman_l:w { l }
5988 \cs_new:Npn \__int_to_roman_c:w { c }
5989 \cs_new:Npn \__int_to_roman_d:w { d }
5990 \cs_new:Npn \__int_to_roman_m:w { m }
5991 \cs_new:Npn \__int_to_roman_Q:w #1 { }
5992 \cs_new:Npn \__int_to_Roman_i:w { I }
5993 \cs_new:Npn \__int_to_Roman_v:w { V }
5994 \cs_new:Npn \__int_to_Roman_x:w { X }
5995 \cs_new:Npn \__int_to_Roman_l:w { L }
5996 \cs_new:Npn \__int_to_Roman_c:w { C }
5997 \cs_new:Npn \__int_to_Roman_d:w { D }
5998 \cs_new:Npn \__int_to_Roman_m:w { M }
5999 \cs_new:Npn \__int_to_Roman_Q:w #1 { }

```

(End definition for `\int_to_roman:n` and others. These functions are documented on page 85.)

10.9 Converting from other formats to integers

```

\__int_pass_signs:wn
\__int_pass_signs_end:wn

```

Called as `__int_pass_signs:wn <signs and digits> \q_stop {<code>}`, this function leaves in the input stream any sign it finds, then inserts the `<code>` before the first non-sign token (and removes `\q_stop`). More precisely, it deletes any `+` and passes any `-` to the input stream, hence should be called in an integer expression.

```

6000 \cs_new:Npn \__int_pass_signs:wn #1
6001 {
6002   \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
6003   \exp_after:wN \__int_pass_signs:wn
6004   \else:
6005     \exp_after:wN \__int_pass_signs_end:wn
6006     \exp_after:wN #1
6007   \fi:
6008 }
6009 \cs_new:Npn \__int_pass_signs_end:wn #1 \q_stop #2 { #2 #1 }

```

(End definition for `__int_pass_signs:wn` and `__int_pass_signs_end:wn`.)

```

\int_from_alph:n
\__int_from_alph:nN
\__int_from_alph:N

```

First take care of signs then loop through the input using the recursion quarks. The `__int_from_alph:nN` auxiliary collects in its first argument the value obtained so far, and the auxiliary `__int_from_alph:N` converts one letter to an expression which evaluates to the correct number.

```

6010 \cs_new:Npn \int_from_alph:n #1
6011 {
6012   \int_eval:n
6013   {
6014     \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
6015     \q_stop { \__int_from_alph:nN { 0 } }
6016     \q_recursion_tail \q_recursion_stop

```

```

6017     }
6018   }
6019   \cs_new:Npn \__int_from_alph:nN #1#2
6020   {
6021     \quark_if_recursion_tail_stop_do:Nn #2 {#1}
6022     \exp_args:Nf \__int_from_alph:nN
6023       { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
6024   }
6025   \cs_new:Npn \__int_from_alph:N #1
6026   { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }

```

(End definition for `\int_from_alph:n`, `__int_from_alph:nN`, and `__int_from_alph:N`. This function is documented on page 85.)

`\int_from_base:nn` Leave the signs into the integer expression, then loop through characters, collecting the value found so far in the first argument of `__int_from_base:nnN`. To convert a single character, `__int_from_base:N` checks first for digits, then distinguishes lower from upper case letters, turning them into the appropriate number. Note that this auxiliary does not use `\int_eval:n`, hence is not safe for general use.

```

6027   \cs_new:Npn \int_from_base:nn #1#2
6028   {
6029     \int_eval:n
6030     {
6031       \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
6032       \q_stop { \__int_from_base:nnN { 0 } {#2} }
6033       \q_recursion_tail \q_recursion_stop
6034     }
6035   }
6036   \cs_new:Npn \__int_from_base:nnN #1#2#3
6037   {
6038     \quark_if_recursion_tail_stop_do:Nn #3 {#1}
6039     \exp_args:Nf \__int_from_base:nnN
6040       { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
6041     {#2}
6042   }
6043   \cs_new:Npn \__int_from_base:N #1
6044   {
6045     \int_compare:nNnTF { '#1 } < { 58 }
6046       {#1}
6047     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
6048   }

```

(End definition for `\int_from_base:nn`, `__int_from_base:nnN`, and `__int_from_base:N`. This function is documented on page 86.)

`\int_from_bin:n` Wrappers around the generic function.

```

\int_from_hex:n
\int_from_oct:n
6049   \cs_new:Npn \int_from_bin:n #1
6050   { \int_from_base:nn {#1} { 2 } }
6051   \cs_new:Npn \int_from_hex:n #1
6052   { \int_from_base:nn {#1} { 16 } }
6053   \cs_new:Npn \int_from_oct:n #1
6054   { \int_from_base:nn {#1} { 8 } }

```

(End definition for `\int_from_bin:n`, `\int_from_hex:n`, and `\int_from_oct:n`. These functions are documented on page 85.)

<code>\c__int_from_roman_i_int</code>	Constants used to convert from Roman numerals to integers.
<code>\c__int_from_roman_v_int</code>	6055 <code>\int_const:cn { c__int_from_roman_i_int } { 1 }</code>
<code>\c__int_from_roman_x_int</code>	6056 <code>\int_const:cn { c__int_from_roman_v_int } { 5 }</code>
<code>\c__int_from_roman_l_int</code>	6057 <code>\int_const:cn { c__int_from_roman_x_int } { 10 }</code>
<code>\c__int_from_roman_c_int</code>	6058 <code>\int_const:cn { c__int_from_roman_l_int } { 50 }</code>
<code>\c__int_from_roman_d_int</code>	6059 <code>\int_const:cn { c__int_from_roman_c_int } { 100 }</code>
<code>\c__int_from_roman_m_int</code>	6060 <code>\int_const:cn { c__int_from_roman_d_int } { 500 }</code>
<code>\c__int_from_roman_I_int</code>	6061 <code>\int_const:cn { c__int_from_roman_m_int } { 1000 }</code>
<code>\c__int_from_roman_V_int</code>	6062 <code>\int_const:cn { c__int_from_roman_I_int } { 1 }</code>
<code>\c__int_from_roman_X_int</code>	6063 <code>\int_const:cn { c__int_from_roman_V_int } { 5 }</code>
<code>\c__int_from_roman_L_int</code>	6064 <code>\int_const:cn { c__int_from_roman_X_int } { 10 }</code>
<code>\c__int_from_roman_C_int</code>	6065 <code>\int_const:cn { c__int_from_roman_L_int } { 50 }</code>
<code>\c__int_from_roman_D_int</code>	6066 <code>\int_const:cn { c__int_from_roman_C_int } { 100 }</code>
<code>\c__int_from_roman_M_int</code>	6067 <code>\int_const:cn { c__int_from_roman_D_int } { 500 }</code>
	6068 <code>\int_const:cn { c__int_from_roman_M_int } { 1000 }</code>

(End definition for `\c__int_from_roman_i_int` and others.)

<code>\int_from_roman:n</code>	The method here is to iterate through the input, finding the appropriate value for each
<code>__int_from_roman:NN</code>	letter and building up a sum. This is then evaluated by \TeX . If any unknown letter is
<code>__int_from_roman_error:w</code>	found, skip to the closing parenthesis and insert <code>*0-1</code> afterwards, to replace the value by
	<code>-1</code> .

```

6069 \cs_new:Npn \int_from_roman:n #1
6070 {
6071   \int_eval:n
6072   {
6073     (
6074       0
6075       \exp_after:wN \__int_from_roman:NN \tl_to_str:n {#1}
6076       \q_recursion_tail \q_recursion_tail \q_recursion_stop
6077     )
6078   }
6079 }
6080 \cs_new:Npn \__int_from_roman:NN #1#2
6081 {
6082   \quark_if_recursion_tail_stop:N #1
6083   \int_if_exist:cF { c__int_from_roman_ #1 _int }
6084   { \__int_from_roman_error:w }
6085   \quark_if_recursion_tail_stop_do:Nn #2
6086   { + \use:c { c__int_from_roman_ #1 _int } }
6087   \int_if_exist:cF { c__int_from_roman_ #2 _int }
6088   { \__int_from_roman_error:w }
6089   \int_compare:nNnTF
6090   { \use:c { c__int_from_roman_ #1 _int } }
6091   <
6092   { \use:c { c__int_from_roman_ #2 _int } }
6093   {
6094     + \use:c { c__int_from_roman_ #2 _int }
6095     - \use:c { c__int_from_roman_ #1 _int }
6096     \__int_from_roman:NN
6097   }
6098   {
6099     + \use:c { c__int_from_roman_ #1 _int }
6100     \__int_from_roman:NN #2

```

```

6101     }
6102   }
6103   \cs_new:Npn \__int_from_roman_error:w #1 \q_recursion_stop #2
6104     { #2 * 0 - 1 }

```

(End definition for `\int_from_roman:n`, `__int_from_roman:NN`, and `__int_from_roman_error:w`. This function is documented on page 86.)

10.10 Viewing integer

`\int_show:N` Diagnostics.
`\int_show:c` 6105 `\cs_new_eq:NN \int_show:N __kernel_register_show:N`
`__int_show:nN` 6106 `\cs_generate_variant:Nn \int_show:N { c }`

(End definition for `\int_show:N` and `__int_show:nN`. This function is documented on page 86.)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output.

```

6107 \cs_new_protected:Npn \int_show:n
6108   { \msg_show_eval:Nn \int_eval:n }

```

(End definition for `\int_show:n`. This function is documented on page 86.)

`\int_log:N` Diagnostics.
`\int_log:c` 6109 `\cs_new_eq:NN \int_log:N __kernel_register_log:N`
6110 `\cs_generate_variant:Nn \int_log:N { c }`

(End definition for `\int_log:N`. This function is documented on page 87.)

`\int_log:n` Similar to `\int_show:n`.
6111 `\cs_new_protected:Npn \int_log:n`
6112 `{ \msg_log_eval:Nn \int_eval:n }`

(End definition for `\int_log:n`. This function is documented on page 87.)

10.11 Random integers

`\int_rand:nn` Defined in `l3fp-random`.

(End definition for `\int_rand:nn`. This function is documented on page 86.)

10.12 Constant integers

`\c_zero` Again, in `l3basics`

(End definition for `\c_zero`. This variable is documented on page 87.)

`\c_one` Low-number values not previously defined.
`\c_two` 6113 `\int_const:Nn \c_one { 1 }`
`\c_three` 6114 `\int_const:Nn \c_two { 2 }`
`\c_four` 6115 `\int_const:Nn \c_three { 3 }`
`\c_five` 6116 `\int_const:Nn \c_four { 4 }`
`\c_six` 6117 `\int_const:Nn \c_five { 5 }`
`\c_seven` 6118 `\int_const:Nn \c_six { 6 }`
`\c_eight` 6119 `\int_const:Nn \c_seven { 7 }`
`\c_nine`
`\c_ten`
`\c_eleven`
`\c_twelve`
`\c_thirteen`
`\c_fourteen`
`\c_fifteen`
`\c_sixteen`

```

6120 \int_const:Nn \c_eight { 8 }
6121 \int_const:Nn \c_nine { 9 }
6122 \int_const:Nn \c_ten { 10 }
6123 \int_const:Nn \c_eleven { 11 }
6124 \int_const:Nn \c_twelve { 12 }
6125 \int_const:Nn \c_thirteen { 13 }
6126 \int_const:Nn \c_fourteen { 14 }
6127 \int_const:Nn \c_fifteen { 15 }
6128 \int_const:Nn \c_sixteen { 16 }

```

(End definition for `\c_one` and others. These variables are documented on page 87.)

`\c_thirty_two` One middling value.

```

6129 \int_const:Nn \c_thirty_two { 32 }

```

(End definition for `\c_thirty_two`. This variable is documented on page 87.)

`\c_two_hundred_fifty_five` Two classic mid-range integer constants.

```

\c_two_hundred_fifty_six 6130 \int_const:Nn \c_two_hundred_fifty_five { 255 }
6131 \int_const:Nn \c_two_hundred_fifty_six { 256 }

```

(End definition for `\c_two_hundred_fifty_five` and `\c_two_hundred_fifty_six`. These variables are documented on page 87.)

`\c_one_hundred` Simple runs of powers of ten.

```

\c_one_thousand 6132 \int_const:Nn \c_one_hundred { 100 }
\c_ten_thousand 6133 \int_const:Nn \c_one_thousand { 1000 }
6134 \int_const:Nn \c_ten_thousand { 10000 }

```

(End definition for `\c_one_hundred`, `\c_one_thousand`, and `\c_ten_thousand`. These variables are documented on page 87.)

`\c_max_int` The largest number allowed is $2^{31} - 1$

```

6135 \int_const:Nn \c_max_int { 2 147 483 647 }

```

(End definition for `\c_max_int`. This variable is documented on page 87.)

`\c_max_char_int` The largest character code is 1114111 (hexadecimal 10FFFF) in X_ƎT_EX and LuaT_EX and 255 in other engines. In many places pT_EX and upT_EX support larger character codes but for instance the values of `\lccode` are restricted to $[0, 255]$.

```

6136 \int_const:Nn \c_max_char_int
6137 {
6138   \if_int_odd:w 0
6139     \cs_if_exist:NT \luatex luatexversion:D { 1 }
6140     \cs_if_exist:NT \xetex XeTeXversion:D { 1 } ~
6141     "10FFFF
6142   \else:
6143     "FF
6144   \fi:
6145 }

```

(End definition for `\c_max_char_int`. This variable is documented on page 87.)

10.13 Scratch integers

`\l_tmpa_int` We provide two local and two global scratch counters, maybe we need more or less.

```

6146 \int_new:N \l_tmpa_int
6147 \int_new:N \l_tmpb_int
6148 \int_new:N \g_tmpa_int
6149 \int_new:N \g_tmpb_int

```

(End definition for `\l_tmpa_int` and others. These variables are documented on page 87.)

10.14 Deprecated

`\c_minus_one` The actual allocation mechanism is in `l3alloc`; it requires `\c_one` to be defined. In package mode, reuse `\m@ne`. We also store in two global token lists some code for `\debug_deprecation_on:` and `\debug_deprecation_off:`. For the latter, we need to locally set `\c_minus_one` back to the constant hence use a private name. We use `\tex_let:D` directly because `\c_minus_one` (as all deprecated commands) is made outer by `\debug_deprecation_on:`.

```

6150 <package>\cs_gset_eq:NN \c__int_minus_one \m@ne
6151 <initex>\int_const:Nn \c__int_minus_one { -1 }
6152 \cs_new_eq:NN \c_minus_one \c__int_minus_one
6153 \__kernel_deprecation_code:nn
6154 { \__kernel_deprecation_error:Nnn \c_minus_one { -1 } { 2018-12-31 } }
6155 { \tex_let:D \c_minus_one \c__int_minus_one }

```

(End definition for `\c_minus_one`.)

`__int_value:w` Made public.

```

6156 \cs_new_eq:NN \__int_value:w \int_value:w

```

(End definition for `__int_value:w`.)

```

6157 </initex | package>

```

11 l3flag implementation

```

6158 <*initex | package>
6159 <@@=flag>

```

The following test files are used for this code: `m3flag001`.

11.1 Non-expandable flag commands

The height h of a flag (initially zero) is stored by setting control sequences of the form `\flag <name> <integer>` to `\relax` for $0 \leq \langle integer \rangle < h$. When a flag is raised, a “trap” function `\flag <name>` is called. The existence of this function is also used to test for the existence of a flag.

`\flag_new:n` For each flag, we define a “trap” function, which by default simply increases the flag by 1 by letting the appropriate control sequence to `\relax`. This can be done expandably!

```

6160 \cs_new_protected:Npn \flag_new:n #1
6161 {
6162   \cs_new:cpn { flag~#1 } ##1 ;
6163   { \exp_after:wN \use_none:n \cs:w flag~#1~##1 \cs_end: }
6164 }

```

(End definition for `\flag_new:n`. This function is documented on page 90.)

`\flag_clear:n` Undefine control sequences, starting from the 0 flag, upwards, until reaching an undefined control sequence. We don't use `\cs_undefine:c` because that would act globally. When the option `check-declarations` is used, check for the function defined by `\flag_new:n`.

`__flag_clear:wn`

```

6165 \__kernel_patch:nnNNpn
6166   { \exp_args:Nc \__kernel_chk_var_exist:N { flag~#1 } } { }
6167 \cs_new_protected:Npn \flag_clear:n #1 { \__flag_clear:wn 0 ; {#1} }
6168 \cs_new_protected:Npn \__flag_clear:wn #1 ; #2
6169   {
6170     \if_cs_exist:w flag~#2~#1 \cs_end:
6171     \cs_set_eq:cN { flag~#2~#1 } \tex_undefined:D
6172     \exp_after:wN \__flag_clear:wn
6173     \int_value:w \int_eval:w 1 + #1
6174   \else:
6175     \use_i:nnn
6176   \fi:
6177   ; {#2}
6178 }

```

(End definition for `\flag_clear:n` and `__flag_clear:wn`. This function is documented on page 90.)

`\flag_clear_new:n` As for other datatypes, clear the `<flag>` or create a new one, as appropriate.

```

6179 \cs_new_protected:Npn \flag_clear_new:n #1
6180   { \flag_if_exist:nTF {#1} { \flag_clear:n } { \flag_new:n } {#1} }

```

(End definition for `\flag_clear_new:n`. This function is documented on page 90.)

`\flag_show:n` Show the height (terminal or log file) using appropriate `l3msg` auxiliaries.
`\flag_log:n`
`__flag_show:Nn`

```

6181 \cs_new_protected:Npn \flag_show:n { \__flag_show:Nn \tl_show:n }
6182 \cs_new_protected:Npn \flag_log:n { \__flag_show:Nn \tl_log:n }
6183 \cs_new_protected:Npn \__flag_show:Nn #1#2
6184   {
6185     \exp_args:Nc \__kernel_chk_defined:NT { flag~#2 }
6186     {
6187       \exp_args:Nx #1
6188       { \tl_to_str:n { flag~#2~height } = \flag_height:n {#2} }
6189     }
6190   }

```

(End definition for `\flag_show:n`, `\flag_log:n`, and `__flag_show:Nn`. These functions are documented on page 90.)

11.2 Expandable flag commands

`__flag_chk_exist:n` Analogue of `__kernel_chk_var_exist:N` for flags, and with an expandable error. We need to add checks by hand because flags are not implemented in terms of other variables. Not all functions need to be patched since some are defined in terms of others.

```

6191 <*package>
6192 \__kernel_if_debug:TF
6193   {
6194     \cs_new:Npn \__flag_chk_exist:n #1
6195     {
6196       \flag_if_exist:nF {#1}

```

```

6197         {
6198             \__kernel_msg_expandable_error:nnn
6199             { kernel } { bad-variable } { flag~#1~ }
6200         }
6201     }
6202 }
6203 { }
6204 \end{package}

```

(End definition for __flag_chk_exist:n.)

\flag_if_exist_p:n A flag exist if the corresponding trap \flag <flag name>:n is defined.

```

\flag_if_exist:nTF
6205 \prg_new_conditional:Npnn \flag_if_exist:n #1 { p , T , F , TF }
6206 {
6207     \cs_if_exist:cTF { flag~#1 }
6208     { \prg_return_true: } { \prg_return_false: }
6209 }

```

(End definition for \flag_if_exist:nTF. This function is documented on page 91.)

\flag_if_raised_p:n Test if the flag has a non-zero height, by checking the 0 control sequence.

```

\flag_if_raised:nTF
6210 \__kernel_patch_conditional:nnNpnn { \__flag_chk_exist:n {#1} }
6211 \prg_new_conditional:Npnn \flag_if_raised:n #1 { p , T , F , TF }
6212 {
6213     \if_cs_exist:w flag~#1~0 \cs_end:
6214     \prg_return_true:
6215     \else:
6216     \prg_return_false:
6217     \fi:
6218 }

```

(End definition for \flag_if_raised:nTF. This function is documented on page 91.)

\flag_height:n Extract the value of the flag by going through all of the control sequences starting from 0.

```

\__flag_height_loop:wn
\__flag_height_end:wn
6219 \__kernel_patch:nnNNpnn { \__flag_chk_exist:n {#1} } { }
6220 \cs_new:Npn \flag_height:n #1 { \__flag_height_loop:wn 0; {#1} }
6221 \cs_new:Npn \__flag_height_loop:wn #1 ; #2
6222 {
6223     \if_cs_exist:w flag~#2~#1 \cs_end:
6224     \exp_after:wN \__flag_height_loop:wn \int_value:w \int_eval:w 1 +
6225     \else:
6226     \exp_after:wN \__flag_height_end:wn
6227     \fi:
6228     #1 ; {#2}
6229 }
6230 \cs_new:Npn \__flag_height_end:wn #1 ; #2 {#1}

```

(End definition for \flag_height:n, __flag_height_loop:wn, and __flag_height_end:wn. This function is documented on page 91.)

\flag_raise:n Simply apply the trap to the height, after expanding the latter.

```

6231 \cs_new:Npn \flag_raise:n #1
6232 {
6233     \cs:w flag~#1 \exp_after:wN \cs_end:
6234     \int_value:w \flag_height:n {#1} ;
6235 }

```

(End definition for `\flag_raise:n`. This function is documented on page 91.)

6236 `</initex | package>`

12 l3prg implementation

The following test files are used for this code: `m3prg001.lvt,m3prg002.lvt,m3prg003.lvt`.

6237 `<*initex | package>`

12.1 Primitive conditionals

`\if_bool:N`
`\if_predicate:w`

Those two primitive TeX conditionals are synonyms.

6238 `\cs_new_eq:NN \if_bool:N \tex_ifodd:D`
6239 `\cs_new_eq:NN \if_predicate:w \tex_ifodd:D`

(End definition for `\if_bool:N` and `\if_predicate:w`. These functions are documented on page 99.)

12.2 Defining a set of conditional functions

These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 92.)

12.3 The boolean data type

6240 `<@@=bool>`

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

6241 `\cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }`
6242 `\cs_generate_variant:Nn \bool_new:N { c }`

(End definition for `\bool_new:N`. This function is documented on page 94.)

`\bool_set_true:N`
`\bool_set_true:c`
`\bool_gset_true:N`
`\bool_gset_true:c`
`\bool_set_false:N`
`\bool_set_false:c`
`\bool_gset_false:N`
`\bool_gset_false:c`

Setting is already pretty easy. When `check-declarations` is active, the definitions are patched to make sure the boolean exists. This is needed because booleans are not based on token lists nor on TeX registers.

6243 `__kernel_patch:nnNNpn { __kernel_chk_var_local:N #1 } { }`
6244 `\cs_new_protected:Npn \bool_set_true:N #1`
6245 `{ \cs_set_eq:NN #1 \c_true_bool }`
6246 `__kernel_patch:nnNNpn { __kernel_chk_var_local:N #1 } { }`
6247 `\cs_new_protected:Npn \bool_set_false:N #1`
6248 `{ \cs_set_eq:NN #1 \c_false_bool }`
6249 `__kernel_patch:nnNNpn { __kernel_chk_var_global:N #1 } { }`
6250 `\cs_new_protected:Npn \bool_gset_true:N #1`
6251 `{ \cs_gset_eq:NN #1 \c_true_bool }`
6252 `__kernel_patch:nnNNpn { __kernel_chk_var_global:N #1 } { }`
6253 `\cs_new_protected:Npn \bool_gset_false:N #1`
6254 `{ \cs_gset_eq:NN #1 \c_false_bool }`
6255 `\cs_generate_variant:Nn \bool_set_true:N { c }`
6256 `\cs_generate_variant:Nn \bool_set_false:N { c }`
6257 `\cs_generate_variant:Nn \bool_gset_true:N { c }`
6258 `\cs_generate_variant:Nn \bool_gset_false:N { c }`

(End definition for `\bool_set_true:N` and others. These functions are documented on page 94.)

`\bool_set_eq:NN` The usual copy code. While it would be cleaner semantically to copy the `\cs_set_eq:NN` family of functions, we copy `\tl_set_eq:NN` because that has the correct checking code.

`\bool_set_eq:cN`

`\bool_set_eq:Nc` 6259 `\cs_new_eq:NN \bool_set_eq:NN \tl_set_eq:NN`

`\bool_set_eq:cc` 6260 `\cs_new_eq:NN \bool_gset_eq:NN \tl_gset_eq:NN`

`\bool_gset_eq:NN` 6261 `\cs_generate_variant:Nn \bool_set_eq:NN { Nc, cN, cc }`

`\bool_gset_eq:cN` 6262 `\cs_generate_variant:Nn \bool_gset_eq:NN { Nc, cN, cc }`

`\bool_gset_eq:Nc` (End definition for `\bool_set_eq:NN` and `\bool_gset_eq:NN`. These functions are documented on page 95.)

`\bool_gset_eq:cc`

`\bool_set:Nn` This function evaluates a boolean expression and assigns the first argument the meaning

`\bool_set:cn` `\c_true_bool` or `\c_false_bool`. Again, we include some checking code.

`\bool_gset:Nn` 6263 `__kernel_patch:nnNNpn { __kernel_chk_var_local:N #1 } { }`

`\bool_gset:cn` 6264 `\cs_new_protected:Npn \bool_set:Nn #1#2`

6265 `{ \tex_chardef:D #1 = \bool_if_p:n {#2} }`

6266 `__kernel_patch:nnNNpn { __kernel_chk_var_global:N #1 } { }`

6267 `\cs_new_protected:Npn \bool_gset:Nn #1#2`

6268 `{ \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }`

6269 `\cs_generate_variant:Nn \bool_set:Nn { c }`

6270 `\cs_generate_variant:Nn \bool_gset:Nn { c }`

(End definition for `\bool_set:Nn` and `\bool_gset:Nn`. These functions are documented on page 95.)

`\bool_if_p:N` Straight forward here. We could optimize here if we wanted to as the boolean can just

`\bool_if_p:c` be input directly.

`\bool_if:NTF` 6271 `\prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }`

`\bool_if:cTF` 6272 `{`

6273 `\if_bool:N #1`

6274 `\prg_return_true:`

6275 `\else:`

6276 `\prg_return_false:`

6277 `\fi:`

6278 `}`

6279 `\prg_generate_conditional_variant:Nnn \bool_if:N { c } { p , T , F , TF }`

(End definition for `\bool_if:NTF`. This function is documented on page 95.)

`\bool_show:n` Show the truth value of the boolean, as true or false.

`\bool_log:n` 6280 `\cs_new_protected:Npn \bool_show:n`

`__bool_to_str:n` 6281 `{ \msg_show_eval:Nn __bool_to_str:n }`

6282 `\cs_new_protected:Npn \bool_log:n`

6283 `{ \msg_log_eval:Nn __bool_to_str:n }`

6284 `\cs_new:Npn __bool_to_str:n #1`

6285 `{ \bool_if:nTF {#1} { true } { false } }`

(End definition for `\bool_show:n`, `\bool_log:n`, and `__bool_to_str:n`. These functions are documented on page 95.)

`\bool_show:N` Show the truth value of the boolean, as true or false.

`\bool_show:c` 6286 `\cs_new_protected:Npn \bool_show:N { __bool_show:NN \tl_show:n }`

`\bool_log:N` 6287 `\cs_generate_variant:Nn \bool_show:N { c }`

`\bool_log:c` 6288 `\cs_new_protected:Npn \bool_log:N { __bool_show:NN \tl_log:n }`

`__bool_show:NN` 6289 `\cs_generate_variant:Nn \bool_log:N { c }`

```

6290 \cs_new_protected:Npn \__bool_show:NN #1#2
6291 {
6292     \__kernel_chk_defined:NT #2
6293     { \exp_args:Nx #1 { \token_to_str:N #2 = \__bool_to_str:n {#2} } }
6294 }

```

(End definition for `\bool_show:N`, `\bool_log:N`, and `__bool_show:NN`. These functions are documented on page 95.)

`\l_tmpa_bool` A few booleans just if you need them.

```

\l_tmpb_bool 6295 \bool_new:N \l_tmpa_bool
\g_tmpa_bool 6296 \bool_new:N \l_tmpb_bool
\g_tmpb_bool 6297 \bool_new:N \g_tmpa_bool
6298 \bool_new:N \g_tmpb_bool

```

(End definition for `\l_tmpa_bool` and others. These variables are documented on page 95.)

`\bool_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

\bool_if_exist_p:c 6299 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N
\bool_if_exist:NTF 6300 { TF , T , F , p }
\bool_if_exist:cTF 6301 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c
6302 { TF , T , F , p }

```

(End definition for `\bool_if_exist:N`. This function is documented on page 95.)

12.4 Boolean expressions

`\bool_if_p:n` Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with `(` and `)` for grouping, `!` for logical “Not”, `&&` for logical “And” and `||` for logical “Or”. However, they perform eager evaluation. We shall use the terms Not, And, Or, Open and Close for these operations.

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a `GetNext` function:

- If an Open is seen, start evaluating a new expression using the `Eval` function and call `GetNext` again.
- If a Not is seen, remove the `!` and call a `GetNext` function with the logic reversed.
- If none of the above, reinsert the token found (this is supposed to be a predicate function) in front of an `Eval` function, which evaluates it to the boolean value $\langle true \rangle$ or $\langle false \rangle$.

The `Eval` function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

$\langle true \rangle$ **And** Current truth value is true, logical And seen, continue with `GetNext` to examine truth value of next boolean (sub-)expression.

$\langle false \rangle$ **And** Current truth value is false, logical And seen, stop using the values of predicates within this sub-expression until the next Close. Then return $\langle false \rangle$.

$\langle true \rangle$ **Or** Current truth value is true, logical Or seen, stop using the values of predicates within this sub-expression until the nearest Close. Then return $\langle true \rangle$.

$\langle false \rangle$ Or Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle true \rangle$ Close Current truth value is true, Close seen, return $\langle true \rangle$.

$\langle false \rangle$ Close Current truth value is false, Close seen, return $\langle false \rangle$.

```

6303 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
6304 {
6305   \if_predicate:w \bool_if_p:n {#1}
6306   \prg_return_true:
6307   \else:
6308   \prg_return_false:
6309   \fi:
6310 }

```

(End definition for `\bool_if:nTF`. This function is documented on page 97.)

`\bool_if_p:n` First issue a `\group_align_safe_begin:` as we are using `&&` as syntax shorthand for the And operation and we need to hide it for $\text{T}_{\text{E}}\text{X}$. This group is closed after `__bool_get_next:NN` returns `\c_true_bool` or `\c_false_bool`. That function requires the trailing parenthesis to know where the expression ends.

```

6311 \cs_new:Npn \bool_if_p:n #1
6312 {
6313   \group_align_safe_begin:
6314   \exp_after:wN
6315   \group_align_safe_end:
6316   \exp:w \exp_end_continue_f:w % (
6317   \__bool_get_next:NN \use_i:nnnn #1 )
6318 }

```

(End definition for `\bool_if_p:n`. This function is documented on page 97.)

`__bool_get_next:NN` The GetNext operation. Its first argument is `\use_i:nnnn`, `\use_ii:nnnn`, `\use_iii:nnnn`, or `\use_iv:nnnn` (we call these “states”). In the first state, this function eventually expand to the truth value `\c_true_bool` or `\c_false_bool` of the expression which follows until the next unmatched closing parenthesis. For instance “`__bool_get_next:NN \use_i:nnnn \c_true_bool && \c_true_bool)`” (including the closing parenthesis) expands to `\c_true_bool`. In the second state (after a `!`) the logic is reversed. We call these two states “normal” and the next two “skipping”. In the third state (after `\c_true_bool||`) it always returns `\c_true_bool`. In the fourth state (after `\c_false_bool&&`) it always returns `\c_false_bool` and also stops when encountering `||`, not only parentheses. This code itself is a switch: if what follows is neither `!` nor `(`, we assume it is a predicate.

```

6319 \cs_new:Npn \__bool_get_next:NN #1#2
6320 {
6321   \use:c
6322   {
6323     __bool_
6324     \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
6325     :Nw
6326   }
6327   #1 #2
6328 }

```

(End definition for _bool_get_next:NN.)

bool!:Nw The Not operation reverses the logic: it discards the ! token and calls the GetNext operation with the appropriate first argument. Namely the first and second states are interchanged, but after \c_true_bool|| or \c_false_bool&& the ! is ignored.

```
6329 \cs_new:cpn { \_bool\_!:Nw } #1#2
6330 {
6331   \exp_after:wN \_bool\_get\_next:NN
6332   #1 \use_ii:nnnn \use_i:nnnn \use_iii:nnnn \use_iv:nnnn
6333 }
```

(End definition for _bool_!:Nw.)

bool(:Nw The Open operation starts a sub-expression after discarding the open parenthesis. This is done by calling GetNext (which eventually discards the corresponding closing parenthesis), with a post-processing step which looks for And, Or or Close after the group.

```
6334 \cs_new:cpn { \_bool\_(:Nw } #1#2
6335 {
6336   \exp_after:wN \_bool\_choose:NNN \exp_after:wN #1
6337   \int_value:w \_bool\_get\_next:NN \use_i:nnnn
6338 }
```

(End definition for _bool_(:Nw.)

_bool_p:Nw If what follows GetNext is neither ! nor (, evaluate the predicate using the primitive \int_value:w. The canonical true and false values have numerical values 1 and 0 respectively. Look for And, Or or Close afterwards.

```
6339 \cs_new:cpn { \_bool\_p:Nw } #1
6340 { \exp_after:wN \_bool\_choose:NNN \exp_after:wN #1 \int_value:w }
```

(End definition for _bool_p:Nw.)

_bool_choose:NNN The arguments are #1: a function such as \use_i:nnnn, #2: 0 or 1 encoding the current truth value, #3: the next operation, And, Or or Close. We distinguish three cases according to a combination of #1 and #2. Case 2 is when #1 is \use_iii:nnnn (state 3), namely after \c_true_bool ||. Case 1 is when #1 is \use_i:nnnn and #2 is true or when #1 is \use_ii:nnnn and #2 is false, for instance for !\c_false_bool. Case 0 includes the same with true/false interchanged and the case where #1 is \use_iv:nnnn namely after \c_false_bool &&.

bool|_0: When seeing) the current subexpression is done, leave the appropriate boolean.
bool|_1: When seeing & in case 0 go into state 4, equivalent to having seen \c_false_bool &&.
bool|_2: In case 1, namely when the argument is true and we are in a normal state continue in the normal state 1. In case 2, namely when skipping alternatives in an Or, continue in the same state. When seeing | in case 0, continue in a normal state; in particular stop skipping for \c_false_bool && because that binds more tightly than ||. In the other two cases start skipping for \c_true_bool ||.

```
6341 \cs_new:Npn \_bool\_choose:NNN #1#2#3
6342 {
6343   \use:c
6344   {
6345     \_bool\_ \token_to_str:N #3 _
6346     #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: } 2 0 :
6347   }
```



```

6348 }
6349 \cs_new:cpn { __bool_)_0: } { \c_false_bool }
6350 \cs_new:cpn { __bool_)_1: } { \c_true_bool }
6351 \cs_new:cpn { __bool_)_2: } { \c_true_bool }
6352 \cs_new:cpn { __bool_&_0: } & { \__bool_get_next:NN \use_iv:nnnn }
6353 \cs_new:cpn { __bool_&_1: } & { \__bool_get_next:NN \use_i:nnnn }
6354 \cs_new:cpn { __bool_&_2: } & { \__bool_get_next:NN \use_iii:nnnn }
6355 \cs_new:cpn { __bool_|_0: } | { \__bool_get_next:NN \use_i:nnnn }
6356 \cs_new:cpn { __bool_|_1: } | { \__bool_get_next:NN \use_iii:nnnn }
6357 \cs_new:cpn { __bool_|_2: } | { \__bool_get_next:NN \use_iii:nnnn }

```

(End definition for __bool_choose:NNN and others.)

\bool_lazy_all_p:n Go through the list of expressions, stopping whenever an expression is **false**. If the end is reached without finding any false expression, then the result is true.

\bool_lazy_all:nTF
__bool_lazy_all:n

```

6358 \prg_new_conditional:Npnn \bool_lazy_all:n #1 { p , T , F , TF }
6359 { \__bool_lazy_all:n #1 \q_recursion_tail \q_recursion_stop }
6360 \cs_new:Npn \__bool_lazy_all:n #1
6361 {
6362   \quark_if_recursion_tail_stop_do:nn {#1} { \prg_return_true: }
6363   \bool_if:nF {#1}
6364   { \use_i_delimit_by_q_recursion_stop:nw { \prg_return_false: } }
6365   \__bool_lazy_all:n
6366 }

```

(End definition for \bool_lazy_all:nTF and __bool_lazy_all:n. This function is documented on page 97.)

\bool_lazy_and_p:nn Only evaluate the second expression if the first is true.

\bool_lazy_and:nnTF

```

6367 \prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
6368 {
6369   \bool_if:nTF {#1}
6370   { \bool_if:nTF {#2} { \prg_return_true: } { \prg_return_false: } }
6371   { \prg_return_false: }
6372 }

```

(End definition for \bool_lazy_and:nnTF. This function is documented on page 97.)

\bool_lazy_any_p:n Go through the list of expressions, stopping whenever an expression is **true**. If the end is reached without finding any true expression, then the result is false.

\bool_lazy_any:nTF
__bool_lazy_any:n

```

6373 \prg_new_conditional:Npnn \bool_lazy_any:n #1 { p , T , F , TF }
6374 { \__bool_lazy_any:n #1 \q_recursion_tail \q_recursion_stop }
6375 \cs_new:Npn \__bool_lazy_any:n #1
6376 {
6377   \quark_if_recursion_tail_stop_do:nn {#1} { \prg_return_false: }
6378   \bool_if:nT {#1}
6379   { \use_i_delimit_by_q_recursion_stop:nw { \prg_return_true: } }
6380   \__bool_lazy_any:n
6381 }

```

(End definition for \bool_lazy_any:nTF and __bool_lazy_any:n. This function is documented on page 97.)

\bool_lazy_or_p:nn Only evaluate the second expression if the first is false.
\bool_lazy_or:nnTF

```

6382 \prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
6383 {
6384   \bool_if:nTF {#1}
6385     { \prg_return_true: }
6386     { \bool_if:nTF {#2} { \prg_return_true: } { \prg_return_false: } }
6387 }

```

(End definition for \bool_lazy_or:nnTF. This function is documented on page 97.)

\bool_not_p:n The Not variant just reverses the outcome of \bool_if_p:n. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```

6388 \cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }

```

(End definition for \bool_not_p:n. This function is documented on page 97.)

\bool_xor_p:nn Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```

6389 \cs_new:Npn \bool_xor_p:nn #1#2
6390 {
6391   \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
6392     \c_false_bool
6393     \c_true_bool
6394 }

```

(End definition for \bool_xor_p:nn. This function is documented on page 98.)

12.5 Logical loops

\bool_while_do:Nn A while loop where the boolean is tested before executing the statement. The “while” version executes the code as long as the boolean is true; the “until” version executes the code as long as the boolean is false.

\bool_while_do:cn
\bool_until_do:Nn
\bool_until_do:cn

```

6395 \cs_new:Npn \bool_while_do:Nn #1#2
6396   { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
6397 \cs_new:Npn \bool_until_do:Nn #1#2
6398   { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
6399 \cs_generate_variant:Nn \bool_while_do:Nn { c }
6400 \cs_generate_variant:Nn \bool_until_do:Nn { c }

```

(End definition for \bool_while_do:Nn and \bool_until_do:Nn. These functions are documented on page 98.)

\bool_do_while:Nn A do-while loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

\bool_do_while:cn
\bool_do_until:Nn
\bool_do_until:cn

```

6401 \cs_new:Npn \bool_do_while:Nn #1#2
6402   { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
6403 \cs_new:Npn \bool_do_until:Nn #1#2
6404   { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
6405 \cs_generate_variant:Nn \bool_do_while:Nn { c }
6406 \cs_generate_variant:Nn \bool_do_until:Nn { c }

```

(End definition for \bool_do_while:Nn and \bool_do_until:Nn. These functions are documented on page 98.)

`\bool_while_do:nn` Loop functions with the test either before or after the first body expansion.

```

\bool_do_while:nn 6407 \cs_new:Npn \bool_while_do:nn #1#2
\bool_until_do:nn 6408 {
\bool_do_until:nn 6409   \bool_if:nT {#1}
6410   {
6411     #2
6412     \bool_while_do:nn {#1} {#2}
6413   }
6414 }
6415 \cs_new:Npn \bool_do_while:nn #1#2
6416 {
6417   #2
6418   \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
6419 }
6420 \cs_new:Npn \bool_until_do:nn #1#2
6421 {
6422   \bool_if:nF {#1}
6423   {
6424     #2
6425     \bool_until_do:nn {#1} {#2}
6426   }
6427 }
6428 \cs_new:Npn \bool_do_until:nn #1#2
6429 {
6430   #2
6431   \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
6432 }

```

(End definition for `\bool_while_do:nn` and others. These functions are documented on page 99.)

12.6 Producing multiple copies

6433 `<@@=prg>`

`\prg_replicate:nn` This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input 25 result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\exp:w`, which ensures that `\prg_replicate:nn` only requires two steps of expansion.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it severely affects the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use, and if necessary, it is possible to write `\prg_replicate:nn {1000} { \prg_replicate:nn {1000} {<code>} }`. An alternative approach is to create a string of m's with `\exp:w` which can be done with just four macros but that method has its own

```

\prg_replicate:nn
  \prg_replicate:N
\__prg_replicate_first:N
  \__prg_replicate_
  \__prg_replicate_0:n
  \__prg_replicate_1:n
  \__prg_replicate_2:n
  \__prg_replicate_3:n
  \__prg_replicate_4:n
  \__prg_replicate_5:n
  \__prg_replicate_6:n
  \__prg_replicate_7:n
  \__prg_replicate_8:n
  \__prg_replicate_9:n
\__prg_replicate_first_~:n
\__prg_replicate_first_0:n
\__prg_replicate_first_1:n
\__prg_replicate_first_2:n
\__prg_replicate_first_3:n
\__prg_replicate_first_4:n
\__prg_replicate_first_5:n
\__prg_replicate_first_6:n
\__prg_replicate_first_7:n
\__prg_replicate_first_8:n
\__prg_replicate_first_9:n

```

problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```

6434 \cs_new:Npn \prg_replicate:nn #1
6435 {
6436   \exp:w
6437     \exp_after:wN \__prg_replicate_first:N
6438     \int_value:w \int_eval:n {#1}
6439     \cs_end:
6440 }
6441 \cs_new:Npn \__prg_replicate:N #1
6442 { \cs:w __prg_replicate_#1 :n \__prg_replicate:N }
6443 \cs_new:Npn \__prg_replicate_first:N #1
6444 { \cs:w __prg_replicate_first_#1 :n \__prg_replicate:N }

```

Then comes all the functions that do the hard work of inserting all the copies. The first function takes :n as a parameter.

```

6445 \cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
6446 \cs_new:cpn { __prg_replicate_0:n } #1
6447 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
6448 \cs_new:cpn { __prg_replicate_1:n } #1
6449 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
6450 \cs_new:cpn { __prg_replicate_2:n } #1
6451 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
6452 \cs_new:cpn { __prg_replicate_3:n } #1
6453 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
6454 \cs_new:cpn { __prg_replicate_4:n } #1
6455 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
6456 \cs_new:cpn { __prg_replicate_5:n } #1
6457 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
6458 \cs_new:cpn { __prg_replicate_6:n } #1
6459 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
6460 \cs_new:cpn { __prg_replicate_7:n } #1
6461 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
6462 \cs_new:cpn { __prg_replicate_8:n } #1
6463 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
6464 \cs_new:cpn { __prg_replicate_9:n } #1
6465 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }

```

Users shouldn't ask for something to be replicated once or even not at all but...

```

6466 \cs_new:cpn { __prg_replicate_first_~:n } #1
6467 {
6468   \exp_end:
6469   \__kernel_msg_expandable_error:nn { kernel } { negative-replication }
6470 }
6471 \cs_new:cpn { __prg_replicate_first_0:n } #1 { \exp_end: }
6472 \cs_new:cpn { __prg_replicate_first_1:n } #1 { \exp_end: #1 }
6473 \cs_new:cpn { __prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
6474 \cs_new:cpn { __prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
6475 \cs_new:cpn { __prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
6476 \cs_new:cpn { __prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
6477 \cs_new:cpn { __prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
6478 \cs_new:cpn { __prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
6479 \cs_new:cpn { __prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
6480 \cs_new:cpn { __prg_replicate_first_9:n } #1
6481 { \exp_end: #1#1#1#1#1#1#1#1#1#1 }

```

(End definition for `\prg_replicate:nn` and others. This function is documented on page 99.)

12.7 Detecting TeX’s mode

`\mode_if_vertical_p:` For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\exp_end:` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```
6482 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
6483 { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_vertical:TF`. This function is documented on page 99.)

`\mode_if_horizontal_p:` For testing horizontal mode.

```
6484 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
6485 { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_horizontal:TF`. This function is documented on page 99.)

`\mode_if_inner_p:` For testing inner mode.

```
6486 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
6487 { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_inner:TF`. This function is documented on page 99.)

`\mode_if_math_p:` For testing math mode. At the beginning of an alignment cell, this should be used only inside a non-expandable function.

```
6488 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
6489 { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_math:TF`. This function is documented on page 99.)

12.8 Internal programming functions

`\group_align_safe_begin:` TeX’s alignment structures present many problems. As Knuth says himself in *TeX: The Program*: “It’s sort of a miracle whenever `\halign` or `\valign` work, [...]” One problem relates to commands that internally issues a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we get some sort of weird error message because the underlying `\futurelet` stores the token at the end of the alignment template. This could be a `&_4` giving a message like `! Misplaced \cr.` or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special group so that TeX still thinks it’s on safe ground but at the same time we don’t want to introduce any brace group that may find its way to the output. The following functions help with this by using code documented only in Appendix D of *The TeXbook*... We place the `\if_false: { \fi:` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```
6490 \cs_new:Npn \group_align_safe_begin:
6491 { \if_int_compare:w \if_false: { \fi: ‘} = \c_zero \fi: }
6492 \cs_new:Npn \group_align_safe_end:
6493 { \if_int_compare:w ‘{ = \c_zero } \fi: }
```

(End definition for `\group_align_safe_begin:` and `\group_align_safe_end:`. These functions are documented on page 101.)

6494 `<@@=prg>`

`\g__kernel_prg_map_int` A nesting counter for mapping.

6495 `\int_new:N \g__kernel_prg_map_int`

(End definition for `\g__kernel_prg_map_int:`.)

`\prg_break_point:Nn` These are defined in `l3basics`, as they are needed “early”. This is just a reminder that is the case!

`\prg_map_break:Nn`

(End definition for `\prg_break_point:Nn` and `\prg_map_break:Nn`. These functions are documented on page 100.)

`\prg_break_point:` Also done in `l3basics` as in format mode these are needed within `l3alloc`.

`\prg_break:`

`\prg_break:n`

(End definition for `\prg_break_point:`, `\prg_break:`, and `\prg_break:n`. These functions are documented on page 100.)

12.9 Deprecated functions

`__prg_break_point:Nn` Made public, but used by a few third-parties. It’s not possible to perfectly support a mixture of `__prg_map_break:Nn` and `\prg_map_break:Nn` because they use different delimiters. The following code only breaks if someone tries to break from two “old-style” `__prg_map_break:Nn` and `\prg_map_break:Nn` mappings in one go. Basically, the `__prg_map_break:Nn` converts a single `__prg_break_point:Nn` to `\prg_break_point:Nn`, and that delimiter had better be the right one. Then we call `\prg_map_break:Nn` which may end up breaking intermediate looks in the (unbraced) argument #1. It is essential to define the `break_point` functions before the corresponding `break` functions: otherwise `\debug_on:n {deprecation} \debug_off:n {deprecation}` would break when trying to restore the definitions because they would involve deprecated commands whose definition has not yet been restored.

```
6496 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \prg_break_point:Nn }
6497 \cs_new:Npn \__prg_break_point:Nn { \prg_break_point:Nn }
6498 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \prg_break_point: }
6499 \cs_new:Npn \__prg_break_point: { \prg_break_point: }
6500 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \prg_map_break:Nn }
6501 \cs_new:Npn \__prg_map_break:Nn #1 \__prg_break_point:Nn
6502   { \prg_map_break:Nn #1 \prg_break_point:Nn }
6503 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \prg_break: }
6504 \cs_new:Npn \__prg_break: #1 \__prg_break_point: { }
6505 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \prg_break:n }
6506 \cs_new:Npn \__prg_break:n #1#2 \__prg_break_point: {#1}
```

(End definition for `__prg_break_point:Nn` and others.)

6507 `</initex | package>`

13 l3sys implementation

```
6508 <*initex | package>
```

```
6509 <@@=sys>
```

13.1 The name of the job

`\c_sys_jobname_str` Inherited from the L^AT_EX3 name for the primitive: this needs to actually contain the text of the job name rather than the name of the primitive, of course.

```
6510 <*initex>
6511 \tex_everyjob:D \exp_after:wN
6512 {
6513   \tex_the:D \tex_everyjob:D
6514   \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
6515 }
6516 </initex>
6517 <*package>
6518 \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
6519 </package>
```

(End definition for `\c_sys_jobname_str`. This variable is documented on page 102.)

13.2 Time and date

`\c_sys_minute_int` Copies of the information provided by T_EX

```
\c_sys_hour_int
\c_sys_day_int
\c_sys_month_int
\c_sys_year_int
6520 \int_const:Nn \c_sys_minute_int
6521 { \int_mod:nn { \tex_time:D } { 60 } }
6522 \int_const:Nn \c_sys_hour_int
6523 { \int_div_truncate:nn { \tex_time:D } { 60 } }
6524 \int_const:Nn \c_sys_day_int { \tex_day:D }
6525 \int_const:Nn \c_sys_month_int { \tex_month:D }
6526 \int_const:Nn \c_sys_year_int { \tex_year:D }
```

(End definition for `\c_sys_minute_int` and others. These variables are documented on page 102.)

13.3 Detecting the engine

`__sys_const:nn` Set the T, F, TF, p forms of #1 to be constants equal to the result of evaluating the boolean expression #2.

```
6527 \cs_new_protected:Npn \__sys_const:nn #1#2
6528 {
6529   \bool_if:nTF {#2}
6530   {
6531     \cs_new_eq:cN { #1 :T } \use:n
6532     \cs_new_eq:cN { #1 :F } \use_none:n
6533     \cs_new_eq:cN { #1 :TF } \use_i:nn
6534     \cs_new_eq:cN { #1 _p: } \c_true_bool
6535   }
6536   {
6537     \cs_new_eq:cN { #1 :T } \use_none:n
6538     \cs_new_eq:cN { #1 :F } \use:n
6539     \cs_new_eq:cN { #1 :TF } \use_ii:nn
6540     \cs_new_eq:cN { #1 _p: } \c_false_bool
6541   }
```

```
6542 }
```

(End definition for `_sys_const:nn`.)

`\sys_if_engine luatex_p:` Set up the engine tests on the basis exactly one test should be true. Mainly a case of looking for the appropriate marker primitive. For `uplTeX`, there is a complexity in that setting `-kanji-internal=sjis` or `-kanji-internal=euc` effective makes it more like `pTeX`. In those cases we therefore report `pTeX` rather than `upTeX`.

```
\sys_if_engine luatex:TF
\sys_if_engine pdftex_p:
\sys_if_engine pdftex:TF
\sys_if_engine ptex_p:
\sys_if_engine ptex:TF
\sys_if_engine uptex_p:
\sys_if_engine uptex:TF
\sys_if_engine xetex_p:
\sys_if_engine xetex:TF
\c_sys_engine_str
6543 \str_const:Nx \c_sys_engine_str
6544 {
6545   \cs_if_exist:NT \luatex luatexversion:D { luatex }
6546   \cs_if_exist:NT \pdftex pdftexversion:D { pdftex }
6547   \cs_if_exist:NT \ptex kanjiskip:D
6548   {
6549     \bool_lazy_and:nnTF
6550     { \cs_if_exist_p:N \uptex disablecjktoken:D }
6551     { \int_compare_p:nNn { \ptex_jis:D "2121 } = { "3000 } }
6552     { uptex }
6553     { ptex }
6554   }
6555   \cs_if_exist:NT \xetex XeTeXversion:D { xetex }
6556 }
6557 \tl_map_inline:nn { { luatex } { pdftex } { ptex } { uptex } { xetex } }
6558 {
6559   \_sys_const:nn { sys_if_engine_ #1 }
6560   { \str_if_eq_x_p:nn \c_sys_engine_str {#1} }
6561 }
```

(End definition for `\sys_if_engine luatex:TF` and others. These functions are documented on page 102.)

13.4 Detecting the output

`\sys_if_output dvi_p:` This is a simple enough concept: the two views here are complementary.

```
\sys_if_output dvi:TF
\sys_if_output pdf_p:
\sys_if_output pdf:TF
\c_sys_output_str
6562 \str_const:Nx \c_sys_output_str
6563 {
6564   \int_compare:nNnTF
6565   { \cs_if_exist_use:NF \pdftex pdfoutput:D { 0 } } > { 0 }
6566   { pdf }
6567   { dvi }
6568 }
6569 \_sys_const:nn { sys_if_output_dvi }
6570 { \str_if_eq_x_p:nn \c_sys_output_str { dvi } }
6571 \_sys_const:nn { sys_if_output_pdf }
6572 { \str_if_eq_x_p:nn \c_sys_output_str { pdf } }
```

(End definition for `\sys_if_output dvi:TF`, `\sys_if_output pdf:TF`, and `\c_sys_output_str`. These functions are documented on page 103.)

```
6573 </initex | package>
```


14 l3clist implementation

The following test files are used for this code: *m3clist002*.

```
6574 (*initex | package)
```

```
6575 (@@=clist)
```

\c_empty_clist An empty comma list is simply an empty token list.

```
6576 \cs_new_eq:NN \c_empty_clist \c_empty_tl
```

(End definition for `\c_empty_clist`. This variable is documented on page 112.)

\l__clist_internal_clist Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before `\clist_new:N`

```
6577 \tl_new:N \l__clist_internal_clist
```

(End definition for `\l__clist_internal_clist`.)

__clist_tmp:w A temporary function for various purposes.

```
6578 \cs_new_protected:Npn \__clist_tmp:w { }
```

(End definition for `__clist_tmp:w`.)

14.1 Removing spaces around items

__clist_trim_next:w Called as `\exp:w __clist_trim_next:w \prg_do_nothing: <comma list>` ... it expands to `{<trimmed item>}` where the `<trimmed item>` is the first non-empty result from removing spaces from both ends of comma-delimited items in the `<comma list>`. The `\prg_do_nothing:` marker avoids losing braces. The test for blank items is a somewhat optimized `\tl_if_empty:oTF` construction; if blank, another item is sought, otherwise trim spaces.

```
6579 \cs_new:Npn \__clist_trim_next:w #1 ,
6580 {
6581   \tl_if_empty:oTF { \use_none:nn #1 ? }
6582   { \__clist_trim_next:w \prg_do_nothing: }
6583   { \tl_trim_spaces_apply:oN {#1} \exp_end: }
6584 }
```

(End definition for `__clist_trim_next:w`.)

__clist_sanitize:n The auxiliary `__clist_sanitize:Nn` receives a delimiter (`\c_empty_tl` the first time, afterwards a comma) and that item as arguments. Unless we are done with the loop it calls `__clist_wrap_item:w` to unbrace the item (using a comma delimiter is safe since #2 came from removing spaces from an argument delimited by a comma) and possibly re-brace it if needed.

```
6585 \cs_new:Npn \__clist_sanitize:n #1
6586 {
6587   \exp_after:wN \__clist_sanitize:Nn \exp_after:wN \c_empty_tl
6588   \exp:w \__clist_trim_next:w \prg_do_nothing:
6589   #1 , \q_recursion_tail , \q_recursion_stop
6590 }
6591 \cs_new:Npn \__clist_sanitize:Nn #1#2
6592 {
6593   \quark_if_recursion_tail_stop:n {#2}
```

```

6594     #1 \__clist_wrap_item:w #2 ,
6595     \exp_after:wN \__clist_sanitize:Nn \exp_after:wN ,
6596     \exp:w \__clist_trim_next:w \prg_do_nothing:
6597 }

```

(End definition for __clist_sanitize:n and __clist_sanitize:Nn.)

__clist_if_wrap:nTF True if the argument must be wrapped to avoid getting altered by some clist operations.
 __clist_if_wrap:w That is the case whenever the argument

- starts or end with a space or contains a comma,
- is empty, or
- consists of a single braced group.

All l3clist functions go through the same test when they need to determine whether to brace an item, so it is not a problem that this test has false positives such as “\q_mark?”. If the argument starts or end with a space or contains a comma then one of the three arguments of __clist_if_wrap:w will have its end delimiter (partly) in one of the three copies of #1 in __clist_if_wrap:nTF; this has a knock-on effect meaning that the result of the expansion is not empty; in that case, wrap. Otherwise, the argument is safe unless it starts with a brace group (or is empty) and it is empty or consists of a single n-type argument.

```

6598 \prg_new_conditional:Npnn \__clist_if_wrap:n #1 { TF }
6599 {
6600   \tl_if_empty:oTF
6601   {
6602     \__clist_if_wrap:w
6603     \q_mark ? #1 ~ \q_mark ? ~ #1 \q_mark , ~ \q_mark #1 ,
6604   }
6605   {
6606     \tl_if_head_is_group:nTF { #1 { } }
6607     {
6608       \tl_if_empty:nTF {#1}
6609       { \prg_return_true: }
6610       {
6611         \tl_if_empty:oTF { \use_none:n #1}
6612         { \prg_return_true: }
6613         { \prg_return_false: }
6614       }
6615     }
6616     { \prg_return_false: }
6617   }
6618   { \prg_return_true: }
6619 }
6620 \cs_new:Npn \__clist_if_wrap:w #1 \q_mark ? ~ #2 ~ \q_mark #3 , { }

```

(End definition for __clist_if_wrap:nTF and __clist_if_wrap:w.)

__clist_wrap_item:w Safe items are put in \exp_not:n, otherwise we put an extra set of braces.

```

6621 \cs_new:Npn \__clist_wrap_item:w #1 ,
6622 { \__clist_if_wrap:nTF {#1} { \exp_not:n { {#1} } } { \exp_not:n {#1} } }

```

(End definition for __clist_wrap_item:w.)

14.2 Allocation and initialisation

\clist_new:N Internally, comma lists are just token lists.

\clist_new:c 6623 \cs_new_eq:NN \clist_new:N \tl_new:N
6624 \cs_new_eq:NN \clist_new:c \tl_new:c

(End definition for \clist_new:N. This function is documented on page 104.)

\clist_const:Nn Creating and initializing a constant comma list is done by sanitizing all items (stripping spaces and braces).

\clist_const:cn
\clist_const:Nx 6625 \cs_new_protected:Npn \clist_const:Nn #1#2
\clist_const:cx 6626 { \tl_const:Nx #1 { _clist_sanitize:n {#2} } }
6627 \cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }

(End definition for \clist_const:Nn. This function is documented on page 105.)

\clist_clear:N Clearing comma lists is just the same as clearing token lists.

\clist_clear:c 6628 \cs_new_eq:NN \clist_clear:N \tl_clear:N
\clist_gclear:N 6629 \cs_new_eq:NN \clist_clear:c \tl_clear:c
\clist_gclear:c 6630 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
6631 \cs_new_eq:NN \clist_gclear:c \tl_gclear:c

(End definition for \clist_clear:N and \clist_gclear:N. These functions are documented on page 105.)

\clist_clear_new:N Once again a copy from the token list functions.

\clist_clear_new:c 6632 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
\clist_gclear_new:N 6633 \cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c
\clist_gclear_new:c 6634 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
6635 \cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c

(End definition for \clist_clear_new:N and \clist_gclear_new:N. These functions are documented on page 105.)

\clist_set_eq:NN Once again, these are simple copies from the token list functions.

\clist_set_eq:cN 6636 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
\clist_set_eq:Nc 6637 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
\clist_set_eq:cc 6638 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
\clist_gset_eq:NN 6639 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
\clist_gset_eq:cN 6640 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\clist_gset_eq:Nc 6641 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\clist_gset_eq:cN 6642 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
\clist_gset_eq:cc 6643 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc

(End definition for \clist_set_eq:NN and \clist_gset_eq:NN. These functions are documented on page 105.)

\clist_set_from_seq:NN Setting a comma list from a comma-separated list is done using a simple mapping. Safe items are put in \exp_not:n, otherwise we put an extra set of braces. The first comma must be removed, except in the case of an empty comma-list.

\clist_set_from_seq:cN
\clist_set_from_seq:Nc
\clist_set_from_seq:cc 6644 \cs_new_protected:Npn \clist_set_from_seq:NN
\clist_gset_from_seq:NN 6645 { _clist_set_from_seq:NNNN \clist_clear:N \tl_set:Nx }
\clist_gset_from_seq:cN 6646 \cs_new_protected:Npn \clist_gset_from_seq:NN
\clist_gset_from_seq:Nc 6647 { _clist_set_from_seq:NNNN \clist_gclear:N \tl_gset:Nx }
\clist_gset_from_seq:cc 6648 \cs_new_protected:Npn _clist_set_from_seq:NNNN #1#2#3#4

_clist_set_from_seq:NNNN
_clist_set_from_seq:n

```

6649 {
6650   \seq_if_empty:NTF #4
6651   { #1 #3 }
6652   {
6653     #2 #3
6654     {
6655       \exp_after:wN \use_none:n \exp:w \exp_end_continue_f:w
6656       \seq_map_function:NN #4 \__clist_set_from_seq:n
6657     }
6658   }
6659 }
6660 \cs_new:Npn \__clist_set_from_seq:n #1
6661 {
6662   ,
6663   \__clist_if_wrap:NTF {#1}
6664   { \exp_not:n { {#1} } }
6665   { \exp_not:n {#1} }
6666 }
6667 \cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }
6668 \cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }
6669 \cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }
6670 \cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }

```

(End definition for `\clist_set_from_seq:NN` and others. These functions are documented on page 105.)

`\clist_concat:NNN` Concatenating comma lists is not quite as easy as it seems, as there needs to be the
`\clist_concat:ccc` correct addition of a comma to the output. So a little work to do.
`\clist_gconcat:NNN`
`\clist_gconcat:ccc`
`__clist_concat:NNNN`

```

6671 \cs_new_protected:Npn \clist_concat:NNN
6672 { \__clist_concat:NNNN \tl_set:Nx }
6673 \cs_new_protected:Npn \clist_gconcat:NNN
6674 { \__clist_concat:NNNN \tl_gset:Nx }
6675 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
6676 {
6677   #1 #2
6678   {
6679     \exp_not:o #3
6680     \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
6681     \exp_not:o #4
6682   }
6683 }
6684 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
6685 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }

```

(End definition for `\clist_concat:NNN`, `\clist_gconcat:NNN`, and `__clist_concat:NNNN`. These functions are documented on page 105.)

`\clist_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.
`\clist_if_exist_p:c`
`\clist_if_exist:NTF`
`\clist_if_exist:cTF`

```

6686 \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N
6687 { TF , T , F , p }
6688 \prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
6689 { TF , T , F , p }

```

(End definition for `\clist_if_exist:NTF`. This function is documented on page 105.)

14.3 Adding data to comma lists

```

\clist_set:Nn
\clist_set:NV
\clist_set:No
\clist_set:Nx
\clist_set:cn
\clist_set:cV
\clist_set:co
\clist_set:cx
\clist_gset:Nn
\clist_gset:NV
\clist_gset:No
\clist_gset:Nx
\clist_gset:cn
\clist_gset:cV
\clist_gset:co
\clist_gset:cx
\clist_put_left:Nn
\clist_put_left:NV
\clist_put_left:No
\clist_put_left:Nx
\clist_put_left:cn
\clist_put_left:cV
\clist_put_left:co
\clist_put_left:cx
\clist_gput_left:Nn
\clist_gput_left:NV
\clist_gput_left:No
\clist_gput_left:Nx
\clist_gput_left:cn
\clist_gput_left:cV
\clist_gput_left:co
\clist_gput_left:cx
\__clist_put_left:NNNn
\clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:cV
\clist_put_right:co
\clist_put_right:cx
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:co
\clist_gput_right:cx
\__clist_put_right:NNNn

```

(End definition for `\clist_set:Nn` and `\clist_gset:Nn`. These functions are documented on page 106.)

Everything is based on concatenation after storing in `\l__clist_internal_clist`. This avoids having to worry here about space-trimming and so on.

```

6690 \cs_new_protected:Npn \clist_set:Nn #1#2
6691 { \tl_set:Nx #1 { \__clist_sanitiz:n {#2} } }
6692 \cs_new_protected:Npn \clist_gset:Nn #1#2
6693 { \tl_gset:Nx #1 { \__clist_sanitiz:n {#2} } }
6694 \cs_generate_variant:Nn \clist_set:Nn { NV , No , Nx , c , cV , co , cx }
6695 \cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }
6696 \cs_new_protected:Npn \clist_put_left:Nn
6697 { \__clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
6698 \cs_new_protected:Npn \clist_gput_left:Nn
6699 { \__clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
6700 \cs_new_protected:Npn \__clist_put_left:NNNn #1#2#3#4
6701 {
6702   #2 \l__clist_internal_clist {#4}
6703   #1 #3 \l__clist_internal_clist #3
6704 }
6705 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
6706 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
6707 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
6708 \cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }

```

(End definition for `\clist_put_left:Nn`, `\clist_gput_left:Nn`, and `__clist_put_left:NNNn`. These functions are documented on page 106.)

```

6709 \cs_new_protected:Npn \clist_put_right:Nn
6710 { \__clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
6711 \cs_new_protected:Npn \clist_gput_right:Nn
6712 { \__clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
6713 \cs_new_protected:Npn \__clist_put_right:NNNn #1#2#3#4
6714 {
6715   #2 \l__clist_internal_clist {#4}
6716   #1 #3 #3 \l__clist_internal_clist
6717 }
6718 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
6719 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
6720 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
6721 \cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }

```

(End definition for `\clist_put_right:Nn`, `\clist_gput_right:Nn`, and `__clist_put_right:NNNn`. These functions are documented on page 106.)

14.4 Comma lists as stacks

```

\clist_get:NN
\clist_get:cn
\__clist_get:wN

```

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma. No need to trim spaces as comma-list *variables* are assumed to have

“cleaned-up” items. (Note that grabbing a comma-delimited item removes an outer pair of braces if present, exactly as needed to uncover the underlying item.)

```

6722 \cs_new_protected:Npn \clist_get:NN #1#2
6723 {
6724   \if_meaning:w #1 \c_empty_clist
6725     \tl_set:Nn #2 { \q_no_value }
6726   \else:
6727     \exp_after:wN \__clist_get:wN #1 , \q_stop #2
6728   \fi:
6729 }
6730 \cs_new_protected:Npn \__clist_get:wN #1 , #2 \q_stop #3
6731 { \tl_set:Nn #3 {#1} }
6732 \cs_generate_variant:Nn \clist_get:NN { c }

```

(End definition for `\clist_get:NN` and `__clist_get:wN`. This function is documented on page 111.)

<code>\clist_pop:NN</code> <code>\clist_pop:cN</code> <code>\clist_gpop:NN</code> <code>\clist_gpop:cN</code> <code>__clist_pop:NNN</code> <code>__clist_pop:wwNNN</code> <code>__clist_pop:wN</code>	An empty clist leads to <code>\q_no_value</code> , otherwise grab until the first comma and assign to the variable. The second argument of <code>__clist_pop:wwNNN</code> is a comma list ending in a comma and <code>\q_mark</code> , unless the original clist contained exactly one item: then the argument is just <code>\q_mark</code> . The next auxiliary picks either <code>\exp_not:n</code> or <code>\use_none:n</code> as #2, ensuring that the result can safely be an empty comma list.
--	---

```

6733 \cs_new_protected:Npn \clist_pop:NN
6734 { \__clist_pop:NNN \tl_set:Nx }
6735 \cs_new_protected:Npn \clist_gpop:NN
6736 { \__clist_pop:NNN \tl_gset:Nx }
6737 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
6738 {
6739   \if_meaning:w #2 \c_empty_clist
6740     \tl_set:Nn #3 { \q_no_value }
6741   \else:
6742     \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
6743   \fi:
6744 }
6745 \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \q_stop #3#4#5
6746 {
6747   \tl_set:Nn #5 {#1}
6748   #3 #4
6749   {
6750     \__clist_pop:wN \prg_do_nothing:
6751     #2 \exp_not:o
6752     , \q_mark \use_none:n
6753     \q_stop
6754   }
6755 }
6756 \cs_new:Npn \__clist_pop:wN #1 , \q_mark #2 #3 \q_stop { #2 {#1} }
6757 \cs_generate_variant:Nn \clist_pop:NN { c }
6758 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for `\clist_pop:NN` and others. These functions are documented on page 111.)

<code>\clist_get:NNTF</code> <code>\clist_get:cNTF</code> <code>\clist_pop:NNTF</code> <code>\clist_pop:cNTF</code> <code>\clist_gpop:NNTF</code> <code>\clist_gpop:cNTF</code> <code>__clist_pop_TF:NNN</code>	The same, as branching code: very similar to the above.
--	---

```

6759 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
6760 {
6761   \if_meaning:w #1 \c_empty_clist

```

```

6762     \prg_return_false:
6763 \else:
6764     \exp_after:wN \__clist_get:wN #1 , \q_stop #2
6765     \prg_return_true:
6766 \fi:
6767 }
6768 \prg_generate_conditional_variant:Nnn \clist_get:NN { c } { T , F , TF }
6769 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
6770 { \__clist_pop_TF:NNN \tl_set:Nx #1 #2 }
6771 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
6772 { \__clist_pop_TF:NNN \tl_gset:Nx #1 #2 }
6773 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
6774 {
6775     \if_meaning:w #2 \c_empty_clist
6776     \prg_return_false:
6777 \else:
6778     \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
6779     \prg_return_true:
6780 \fi:
6781 }
6782 \prg_generate_conditional_variant:Nnn \clist_pop:NN { c } { T , F , TF }
6783 \prg_generate_conditional_variant:Nnn \clist_gpop:NN { c } { T , F , TF }

```

(End definition for \clist_get:NNTF and others. These functions are documented on page 111.)

\clist_push:Nn Pushing to a comma list is the same as adding on the left.

\clist_push:Nv	6784 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
\clist_push:No	6785 \cs_new_eq:NN \clist_push:Nv \clist_put_left:Nv
\clist_push:Nx	6786 \cs_new_eq:NN \clist_push:No \clist_put_left:No
\clist_push:cn	6787 \cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx
\clist_push:cV	6788 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn
\clist_push:co	6789 \cs_new_eq:NN \clist_push:cV \clist_put_left:cV
\clist_push:cx	6790 \cs_new_eq:NN \clist_push:co \clist_put_left:co
\clist_gpush:Nn	6791 \cs_new_eq:NN \clist_push:cx \clist_put_left:cx
\clist_gpush:Nv	6792 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\clist_gpush:No	6793 \cs_new_eq:NN \clist_gpush:Nv \clist_gput_left:Nv
\clist_gpush:Nx	6794 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
\clist_gpush:cn	6795 \cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
\clist_gpush:cV	6796 \cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
\clist_gpush:co	6797 \cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
\clist_gpush:cx	6798 \cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
	6799 \cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx

(End definition for \clist_push:Nn and \clist_gpush:Nn. These functions are documented on page 111.)

14.5 Modifying comma lists

\l__clist_internal_remove_clist An internal comma list and a sequence for the removal routines.

```

\l__clist_internal_remove_seq
6800 \clist_new:N \l__clist_internal_remove_clist
6801 \seq_new:N \l__clist_internal_remove_seq

```

(End definition for \l__clist_internal_remove_clist and \l__clist_internal_remove_seq.)

`\clist_remove_duplicates:N` Removing duplicates means making a new list then copying it.

```

\clist_remove_duplicates:c 6802 \cs_new_protected:Npn \clist_remove_duplicates:N
\clist_gremove_duplicates:N 6803 { \__clist_remove_duplicates:NN \clist_set_eq:NN }
\clist_gremove_duplicates:c 6804 \cs_new_protected:Npn \clist_gremove_duplicates:N
    \__clist_remove_duplicates:NN 6805 { \__clist_remove_duplicates:NN \clist_gset_eq:NN }
    \__clist_remove_duplicates:NN 6806 \cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2
    {
6807
6808     \clist_clear:N \l__clist_internal_remove_clist
6809     \clist_map_inline:Nn #2
6810     {
6811         \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
6812         { \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }
6813     }
6814     #1 #2 \l__clist_internal_remove_clist
6815 }
6816 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
6817 \cs_generate_variant:Nn \clist_gremove_duplicates:N { c }

```

(End definition for `\clist_remove_duplicates:N`, `\clist_gremove_duplicates:N`, and `__clist_remove_duplicates:NN`. These functions are documented on page 107.)

`\clist_remove_all:Nn` The method used here for safe items is very similar to `\tl_replace_all:Nnn`. However, if the item contains commas or leading/trailing spaces, or is empty, or consists of a single brace group, we know that it can only appear within braces so the code would fail; instead just convert to a sequence and do the removal with `l3seq` code (it involves somewhat elaborate code to do most of the work expandably but the final token list comparisons non-expandably).

`\clist_remove_all:cn`

`\clist_gremove_all:Nn`

`\clist_gremove_all:cn`

`__clist_remove_all:NNNn`

`__clist_remove_all:w`

`__clist_remove_all:`

For “safe” items, build a function delimited by the `<item>` that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the `<item>`. The loop is controlled by the argument grabbed by `__clist_remove_all:w`: when the item was found, the `\q_mark` delimiter used is the one inserted by `__clist_tmp:w`, and `\use_none_delimit_by_q_stop:w` is deleted. At the end, the final `<item>` is grabbed, and the argument of `__clist_tmp:w` contains `\q_mark`: in that case, `__clist_remove_all:w` removes the second `\q_mark` (inserted by `__clist_tmp:w`), and lets `\use_none_delimit_by_q_stop:w` act.

No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn’t remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.

```

6818 \cs_new_protected:Npn \clist_remove_all:Nn
6819 { \__clist_remove_all:NNNn \clist_set_from_seq:NN \tl_set:Nx }
6820 \cs_new_protected:Npn \clist_gremove_all:Nn
6821 { \__clist_remove_all:NNNn \clist_gset_from_seq:NN \tl_gset:Nx }
6822 \cs_new_protected:Npn \__clist_remove_all:NNNn #1#2#3#4
6823 {
6824     \__clist_if_wrap:nTF {#4}
6825     {
6826         \seq_set_from_clist:NN \l__clist_internal_remove_seq #3
6827         \seq_remove_all:Nn \l__clist_internal_remove_seq {#4}
6828         #1 #3 \l__clist_internal_remove_seq
6829     }
6830     {

```



```

6831 \cs_set:Npn \__clist_tmp:w ##1 , #4 ,
6832 {
6833   ##1
6834   , \q_mark , \use_none_delimit_by_q_stop:w ,
6835   \__clist_remove_all:
6836 }
6837 #2 #3
6838 {
6839   \exp_after:wN \__clist_remove_all:
6840   #3 , \q_mark , #4 , \q_stop
6841 }
6842 \clist_if_empty:NF #3
6843 {
6844   #2 #3
6845   {
6846     \exp_args:No \exp_not:o
6847     { \exp_after:wN \use_none:n #3 }
6848   }
6849 }
6850 }
6851 }
6852 \cs_new:Npn \__clist_remove_all:
6853 { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
6854 \cs_new:Npn \__clist_remove_all:w #1 , \q_mark , #2 , { \exp_not:n {#1} }
6855 \cs_generate_variant:Nn \clist_remove_all:Nn { c }
6856 \cs_generate_variant:Nn \clist_gremove_all:Nn { c }

```

(End definition for `\clist_remove_all:Nn` and others. These functions are documented on page 107.)

`\clist_reverse:N` Use `\clist_reverse:n` in an x-expanding assignment. The extra work that `\clist_reverse:n` does to preserve braces and spaces would not be needed for the well-controlled case of N-type comma lists, but the slow-down is not too bad.

`\clist_reverse:c`

`\clist_greverse:N`

`\clist_greverse:c`

```

6857 \cs_new_protected:Npn \clist_reverse:N #1
6858 { \tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
6859 \cs_new_protected:Npn \clist_greverse:N #1
6860 { \tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
6861 \cs_generate_variant:Nn \clist_reverse:N { c }
6862 \cs_generate_variant:Nn \clist_greverse:N { c }

```

(End definition for `\clist_reverse:N` and `\clist_greverse:N`. These functions are documented on page 107.)

`\clist_reverse:n` The reversed token list is built one item at a time, and stored between `\q_stop` and `\q_mark`, in the form of ? followed by zero or more instances of “ $\langle item \rangle$,”. We start from a comma list “ $\langle item_1 \rangle, \dots, \langle item_n \rangle$ ”. During the loop, the auxiliary `__clist_reverse:wwNww` receives “ $\langle item_i \rangle$ ” as #1, “ $\langle item_{i+1} \rangle, \dots, \langle item_n \rangle$ ” as #2, `__clist_reverse:wwNww` as #3, what remains until `\q_stop` as #4, and “ $\langle item_{i-1} \rangle, \dots, \langle item_1 \rangle$,” as #5. The auxiliary moves #1 just before #5, with a comma, and calls itself (#3). After the last item is moved, `__clist_reverse:wwNww` receives “`\q_mark __clist_reverse:wwNww !`” as its argument #1, thus `__clist_reverse_end:ww` as its argument #3. This second auxiliary cleans up until the marker !, removes the trailing comma (introduced when the first item was moved after `\q_stop`), and leaves its argument #1 within `\exp_not:n`. There is also a need to remove a leading comma, hence `\exp_not:o` and `\use_none:n`.

```

6863 \cs_new:Npn \clist_reverse:n #1
6864 {
6865   \__clist_reverse:wwNww ? #1 ,
6866   \q_mark \__clist_reverse:wwNww ! ,
6867   \q_mark \__clist_reverse_end:ww
6868   \q_stop ? \q_mark
6869 }
6870 \cs_new:Npn \__clist_reverse:wwNww
6871   #1 , #2 \q_mark #3 #4 \q_stop ? #5 \q_mark
6872   { #3 ? #2 \q_mark #3 #4 \q_stop #1 , #5 \q_mark }
6873 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \q_mark
6874   { \exp_not:o { \use_none:n #2 } }

```

(End definition for `\clist_reverse:n`, `__clist_reverse:wwNww`, and `__clist_reverse_end:ww`. This function is documented on page 107.)

`\clist_sort:Nn` Implemented in `l3sort`.

`\clist_sort:cn`

`\clist_gsort:Nn` (End definition for `\clist_sort:Nn` and `\clist_gsort:Nn`. These functions are documented on page 107.)

`\clist_gsort:cn`

14.6 Comma list conditionals

`\clist_if_empty_p:N` Simple copies from the token list variable material.

`\clist_if_empty_p:c`

`\clist_if_empty:NTF`

`\clist_if_empty:cTF`

```

6875 \prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N
6876   { p , T , F , TF }
6877 \prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c
6878   { p , T , F , TF }

```

(End definition for `\clist_if_empty:NTF`. This function is documented on page 107.)

`\clist_if_empty_p:n`

`\clist_if_empty:nTF`

`__clist_if_empty_n:w`

`__clist_if_empty_n:wNw`

As usual, we insert a token (here `?`) before grabbing any argument: this avoids losing braces. The argument of `\tl_if_empty:oTF` is empty if `#1` is `?` followed by blank spaces (besides, this particular variant of the emptiness test is optimized). If the item of the comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second auxiliary grabs `\prg_return_false:` as `#2`, unless every item in the comma list was blank and the loop actually got broken by the trailing `\q_mark \prg_return_false:` item.

```

6879 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
6880 {
6881   \__clist_if_empty_n:w ? #1
6882   , \q_mark \prg_return_false:
6883   , \q_mark \prg_return_true:
6884   \q_stop
6885 }
6886 \cs_new:Npn \__clist_if_empty_n:w #1 ,
6887 {
6888   \tl_if_empty:oTF { \use_none:nn #1 ? }
6889   { \__clist_if_empty_n:w ? }
6890   { \__clist_if_empty_n:wNw }
6891 }
6892 \cs_new:Npn \__clist_if_empty_n:wNw #1 \q_mark #2#3 \q_stop {#2}

```

(End definition for `\clist_if_empty:nTF`, `__clist_if_empty_n:w`, and `__clist_if_empty_n:wNw`. This function is documented on page 108.)

`\clist_if_in:NnTF` For “safe” items, we simply surround the comma list, and the item, with commas, then use the same code as for `\tl_if_in:Nn`. For “unsafe” items we follow the same route as `\seq_if_in:Nn`, mapping through the list a comparison function. If found, return true and remove `\prg_return_false:`.

```

6893 \prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }
6894 {
6895   \exp_args:No \__clist_if_in_return:nnN #1 {#2} #1
6896 }
6897 \prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }
6898 {
6899   \clist_set:Nn \l__clist_internal_clist {#1}
6900   \exp_args:No \__clist_if_in_return:nnN \l__clist_internal_clist {#2}
6901   \l__clist_internal_clist
6902 }
6903 \cs_new_protected:Npn \__clist_if_in_return:nnN #1#2#3
6904 {
6905   \__clist_if_wrap:nTF {#2}
6906   {
6907     \cs_set:Npx \__clist_tmp:w ##1
6908     {
6909       \exp_not:N \tl_if_eq:nnT {##1}
6910       \exp_not:n
6911       {
6912         {#2}
6913         { \clist_map_break:n { \prg_return_true: \use_none:n } }
6914       }
6915     }
6916     \clist_map_function:NN #3 \__clist_tmp:w
6917     \prg_return_false:
6918   }
6919   {
6920     \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
6921     \tl_if_empty:oTF
6922     { \__clist_tmp:w ,#1, { } { } ,#2, }
6923     { \prg_return_false: } { \prg_return_true: }
6924   }
6925 }
6926 \prg_generate_conditional_variant:Nnn \clist_if_in:Nn
6927 { NV , No , c , cV , co } { T , F , TF }
6928 \prg_generate_conditional_variant:Nnn \clist_if_in:nn
6929 { nV , no } { T , F , TF }

```

(End definition for `\clist_if_in:NnTF`, `\clist_if_in:nnTF`, and `__clist_if_in_return:nnN`. These functions are documented on page 108.)

14.7 Mapping to comma lists

`\clist_map_function:NN` If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing one comma-delimited item at a time. The end is marked by `\q_recursion_tail`. The auxiliary function `__clist_map_function:Nw` is also used in `\clist_map_inline:Nn`.

```

6930 \cs_new:Npn \clist_map_function:NN #1#2
6931 {

```

```

6932 \clist_if_empty:NF #1
6933 {
6934   \exp_last_unbraced:NNo \__clist_map_function:Nw #2 #1
6935   , \q_recursion_tail ,
6936   \prg_break_point:Nn \clist_map_break: { }
6937 }
6938 }
6939 \cs_new:Npn \__clist_map_function:Nw #1#2 ,
6940 {
6941   \quark_if_recursion_tail_break:nN {#2} \clist_map_break:
6942   #1 {#2}
6943   \__clist_map_function:Nw #1
6944 }
6945 \cs_generate_variant:Nn \clist_map_function:NN { c }

```

(End definition for `\clist_map_function:NN` and `__clist_map_function:Nw`. This function is documented on page 108.)

`\clist_map_function:nN`
`__clist_map_function_n:Nn`
`__clist_map_unbrace:Nw`

The `n`-type mapping function is a bit more awkward, since spaces must be trimmed from each item. Space trimming is again based on `__clist_trim_next:w`. The auxiliary `__clist_map_function_n:Nn` receives as arguments the function, and the next non-empty item (after space trimming but before brace removal). One level of braces is removed by `__clist_map_unbrace:Nw`.

```

6946 \cs_new:Npn \clist_map_function:nN #1#2
6947 {
6948   \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #2
6949   \exp:w \__clist_trim_next:w \prg_do_nothing: #1 , \q_recursion_tail ,
6950   \prg_break_point:Nn \clist_map_break: { }
6951 }
6952 \cs_new:Npn \__clist_map_function_n:Nn #1 #2
6953 {
6954   \quark_if_recursion_tail_break:nN {#2} \clist_map_break:
6955   \__clist_map_unbrace:Nw #1 #2,
6956   \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #1
6957   \exp:w \__clist_trim_next:w \prg_do_nothing:
6958 }
6959 \cs_new:Npn \__clist_map_unbrace:Nw #1 #2, { #1 {#2} }

```

(End definition for `\clist_map_function:nN`, `__clist_map_function_n:Nn`, and `__clist_map_unbrace:Nw`. This function is documented on page 108.)

`\clist_map_inline:nN`
`\clist_map_inline:cn`
`\clist_map_inline:nn`

Inline mapping is done by creating a suitable function “on the fly”: this is done globally to avoid any issues with TeX’s groups. We use a different function for each level of nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the `n` version simply by storing the comma-list in a variable. We don’t need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```

6960 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
6961 {
6962   \clist_if_empty:NF #1
6963   {
6964     \int_gincr:N \g__kernel_prg_map_int
6965     \cs_gset_protected:cpn
6966     { __clist_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}

```

```

6967     \exp_last_unbraced:Nco \__clist_map_function:Nw
6968     { \__clist_map_ \int_use:N \g__kernel_prg_map_int :w }
6969     #1 , \q_recursion_tail ,
6970     \prg_break_point:Nn \clist_map_break:
6971     { \int_gdecr:N \g__kernel_prg_map_int }
6972   }
6973 }
6974 \cs_new_protected:Npn \clist_map_inline:nn #1
6975 {
6976   \clist_set:Nn \l__clist_internal_clist {#1}
6977   \clist_map_inline:Nn \l__clist_internal_clist
6978 }
6979 \cs_generate_variant:Nn \clist_map_inline:Nn { c }

```

(End definition for `\clist_map_inline:Nn` and `\clist_map_inline:nn`. These functions are documented on page 108.)

`\clist_map_variable:NNn` As for other comma-list mappings, filter out the case of an empty list. Same approach
`\clist_map_variable:cNn` as `\clist_map_function:Nn`, additionally we store each item in the given variable. As
`\clist_map_variable:nNn` for inline mappings, space trimming for the `n` variant is done by storing the comma
`__clist_map_variable:Nnw` list in a variable. The strange `\use:n` avoids unlikely problems when `#2` would contain
`\q_recursion_stop`.

```

6980 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
6981 {
6982   \clist_if_empty:NF #1
6983   {
6984     \exp_args:Nno \use:nn
6985     { \__clist_map_variable:Nnw #2 {#3} }
6986     #1
6987     , \q_recursion_tail , \q_recursion_stop
6988     \prg_break_point:Nn \clist_map_break: { }
6989   }
6990 }
6991 \cs_new_protected:Npn \clist_map_variable:nNn #1
6992 {
6993   \clist_set:Nn \l__clist_internal_clist {#1}
6994   \clist_map_variable:NNn \l__clist_internal_clist
6995 }
6996 \cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
6997 {
6998   \tl_set:Nn #1 {#3}
6999   \quark_if_recursion_tail_stop:N #1
7000   \use:n {#2}
7001   \__clist_map_variable:Nnw #1 {#2}
7002 }
7003 \cs_generate_variant:Nn \clist_map_variable:NNn { c }

```

(End definition for `\clist_map_variable:NNn`, `\clist_map_variable:nNn`, and `__clist_map_variable:Nnw`. These functions are documented on page 109.)

`\clist_map_break:` The break statements use the general `\prg_map_break:Nn` mechanism.

```

\clist_map_break:n
7004 \cs_new:Npn \clist_map_break:
7005 { \prg_map_break:Nn \clist_map_break: { } }
7006 \cs_new:Npn \clist_map_break:n
7007 { \prg_map_break:Nn \clist_map_break: }

```

(End definition for `\clist_map_break:` and `\clist_map_break:n`. These functions are documented on page 109.)

`\clist_count:N` Counting the items in a comma list is done using the same approach as for other token count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics. In the case of an n-type comma-list, we could of course use `\clist_map_function:nN`, but that is very slow, because it carefully removes spaces. Instead, we loop manually, and skip blank items (but not {}, hence the extra spaces).

```

7008 \cs_new:Npn \clist_count:N #1
7009 {
7010   \int_eval:n
7011   {
7012     0
7013     \clist_map_function:NN #1 \__clist_count:n
7014   }
7015 }
7016 \cs_generate_variant:Nn \clist_count:N { c }
7017 \cs_new:Npx \clist_count:n #1
7018 {
7019   \exp_not:N \int_eval:n
7020   {
7021     0
7022     \exp_not:N \__clist_count:w \c_space_tl
7023     #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
7024   }
7025 }
7026 \cs_new:Npn \__clist_count:n #1 { + 1 }
7027 \cs_new:Npx \__clist_count:w #1 ,
7028 {
7029   \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
7030   \exp_not:N \tl_if_blank:nF {#1} { + 1 }
7031   \exp_not:N \__clist_count:w \c_space_tl
7032 }

```

(End definition for `\clist_count:N` and others. These functions are documented on page 109.)

14.8 Using comma lists

`\clist_use:Nnnn` First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the *separator between two* in the middle.

`__clist_use:nwwwnwn` Otherwise, `__clist_use:nwwwnwn` takes the following arguments; 1: a *separator*, 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6: a *continuation* function (`use_ii` or `use_iii` with its *separator* argument), 7: junk, and 8: the temporary result, which is built in a brace group following `\q_stop`. The *separator* and the first of the three items are placed in the result, then we use the *continuation*, placing the remaining two items after it. When we begin this loop, the three items really belong to the comma list, the first `\q_mark` is taken as a delimiter to the `use_ii` function, and the continuation is `use_ii` itself. When we reach the last two items of the original token list, `\q_mark` is taken as a third item, and now the second

`\q_mark` serves as a delimiter to `use_ii`, switching to the other *⟨continuation⟩*, `use_iii`, which uses the *⟨separator between final two⟩*.

```

7033 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
7034 {
7035   \clist_if_exist:NTF #1
7036   {
7037     \int_case:nnF { \clist_count:N #1 }
7038     {
7039       { 0 } { }
7040       { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
7041       { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
7042     }
7043     {
7044       \exp_after:wN \__clist_use:nwwwwnwn
7045       \exp_after:wN { \exp_after:wN } #1 ,
7046       \q_mark , { \__clist_use:nwwwwnwn {#3} }
7047       \q_mark , { \__clist_use:nwnn {#4} }
7048       \q_stop { }
7049     }
7050   }
7051   {
7052     \__kernel_msg_expandable_error:nnn
7053     { kernel } { bad-variable } {#1}
7054   }
7055 }
7056 \cs_generate_variant:Nn \clist_use:Nnnn { c }
7057 \cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
7058 \cs_new:Npn \__clist_use:nwwwwnwn
7059   #1#2 , #3 , #4 , #5 \q_mark , #6#7 \q_stop #8
7060   { #6 {#3} , {#4} , #5 \q_mark , {#6} #7 \q_stop { #8 #1 #2 } }
7061 \cs_new:Npn \__clist_use:nwnn #1#2 , #3 \q_stop #4
7062   { \exp_not:n { #4 #1 #2 } }
7063 \cs_new:Npn \clist_use:Nn #1#2
7064   { \clist_use:Nnnn #1 {#2} {#2} {#2} }
7065 \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End definition for `\clist_use:Nnnn` and others. These functions are documented on page [110](#).)

14.9 Using a single item

`\clist_item:Nn` To avoid needing to test the end of the list at each step, we first compute the *⟨length⟩* of the list. If the item number is 0, less than $-\langle length \rangle$, or more than $\langle length \rangle$, the result is empty. If it is negative, but not less than $-\langle length \rangle$, add $\langle length \rangle + 1$ to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat.

```

\__clist_item:nnnN
\__clist_item:ffoN
\__clist_item:ffnN
\__clist_item_N_loop:nw
7066 \cs_new:Npn \clist_item:Nn #1#2
7067 {
7068   \__clist_item:ffoN
7069   { \clist_count:N #1 }
7070   { \int_eval:n {#2} }
7071   #1
7072   \__clist_item_N_loop:nw
7073 }

```

```

7074 \cs_new:Npn \__clist_item:nnnN #1#2#3#4
7075 {
7076   \int_compare:nNnTF {#2} < 0
7077   {
7078     \int_compare:nNnTF {#2} < { - #1 }
7079     { \use_none_delimit_by_q_stop:w }
7080     { \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } }
7081   }
7082   {
7083     \int_compare:nNnTF {#2} > {#1}
7084     { \use_none_delimit_by_q_stop:w }
7085     { #4 {#2} }
7086   }
7087   { } , #3 , \q_stop
7088 }
7089 \cs_generate_variant:Nn \__clist_item:nnnN { ffo, ff }
7090 \cs_new:Npn \__clist_item_N_loop:nw #1 #2,
7091 {
7092   \int_compare:nNnTF {#1} = 0
7093   { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }
7094   { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
7095 }
7096 \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for `\clist_item:Nn`, `__clist_item:nnnN`, and `__clist_item_N_loop:nw`. This function is documented on page [112](#).)

<pre> \clist_item:nn __clist_item_n:nw __clist_item_n_loop:nw __clist_item_n_end:n __clist_item_n_strip:n __clist_item_n_strip:w </pre>	<p>This starts in the same way as <code>\clist_item:Nn</code> by counting the items of the comma list. The final item should be space-trimmed before being brace-stripped, hence we insert a couple of odd-looking <code>\prg_do_nothing:</code> to avoid losing braces. Blank items are ignored.</p> <pre> 7097 \cs_new:Npn \clist_item:nn #1#2 7098 { 7099 __clist_item:ffnN 7100 { \clist_count:n {#1} } 7101 { \int_eval:n {#2} } 7102 {#1} 7103 __clist_item_n:nw 7104 } 7105 \cs_new:Npn __clist_item_n:nw #1 7106 { __clist_item_n_loop:nw {#1} \prg_do_nothing: } 7107 \cs_new:Npn __clist_item_n_loop:nw #1 #2, 7108 { 7109 \exp_args:No \tl_if_blank:nTF {#2} 7110 { __clist_item_n_loop:nw {#1} \prg_do_nothing: } 7111 { 7112 \int_compare:nNnTF {#1} = 0 7113 { \exp_args:No __clist_item_n_end:n {#2} } 7114 { 7115 \exp_args:Nf __clist_item_n_loop:nw 7116 { \int_eval:n { #1 - 1 } } 7117 \prg_do_nothing: 7118 } 7119 } 7120 } </pre>
--	---


```

7121 \cs_new:Npn \__clist_item_n_end:n #1 #2 \q_stop
7122 { \tl_trim_spaces_apply:nN {#1} \__clist_item_n_strip:n }
7123 \cs_new:Npn \__clist_item_n_strip:n #1 { \__clist_item_n_strip:w #1 , }
7124 \cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }

```

(End definition for `\clist_item:nn` and others. This function is documented on page 112.)

14.10 Viewing comma lists

```

\clist_show:N Apply the general \__kernel_chk_defined:NT and \msg_show:nnnnnn.
\clist_show:c 7125 \cs_new_protected:Npn \clist_show:N { \__clist_show:NN \msg_show:nnxxxx }
\clist_log:N   7126 \cs_generate_variant:Nn \clist_show:N { c }
\clist_log:c   7127 \cs_new_protected:Npn \clist_log:N { \__clist_show:NN \msg_log:nnxxxx }
\__clist_show:NN 7128 \cs_generate_variant:Nn \clist_log:N { c }
                7129 \cs_new_protected:Npn \__clist_show:NN #1#2
                7130 {
                7131     \__kernel_chk_defined:NT #2
                7132     {
                7133         #1 { LaTeX/kernel } { show-clist }
                7134         { \token_to_str:N #2 }
                7135         { \clist_map_function:NN #2 \msg_show_item:n }
                7136         { } { }
                7137     }
                7138 }

```

(End definition for `\clist_show:N`, `\clist_log:N`, and `__clist_show:NN`. These functions are documented on page 112.)

```

\clist_show:n A variant of the above: no existence check, empty first argument for the message.
\clist_log:n   7139 \cs_new_protected:Npn \clist_show:n { \__clist_show:Nn \msg_show:nnxxxx }
\__clist_show:Nn 7140 \cs_new_protected:Npn \clist_log:n { \__clist_show:Nn \msg_log:nnxxxx }
                7141 \cs_new_protected:Npn \__clist_show:Nn #1#2
                7142 {
                7143     #1 { LaTeX/kernel } { show-clist }
                7144     { } { \clist_map_function:nN {#2} \msg_show_item:n } { } { }
                7145 }

```

(End definition for `\clist_show:n`, `\clist_log:n`, and `__clist_show:Nn`. These functions are documented on page 112.)

14.11 Scratch comma lists

```

\l_tmpa_clist Temporary comma list variables.
\l_tmpb_clist 7146 \clist_new:N \l_tmpa_clist
\g_tmpa_clist 7147 \clist_new:N \l_tmpb_clist
\g_tmpb_clist 7148 \clist_new:N \g_tmpa_clist
              7149 \clist_new:N \g_tmpb_clist

```

(End definition for `\l_tmpa_clist` and others. These variables are documented on page 112.)

```

7150 </initex | package>

```

15 l3token implementation

```
7151 \(*initex | package)
```

```
7152 \@@=char
```

15.1 Manipulating and interrogating character tokens

Simple wrappers around the primitives.

```
\char_set_catcode:nn
\char_value_catcode:n
\char_show_value_catcode:n

7153 \cs_new_protected:Npn \char_set_catcode:nn #1#2
7154 { \tex_catcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
7155 \cs_new:Npn \char_value_catcode:n #1
7156 { \tex_the:D \tex_catcode:D \int_eval:n {#1} \exp_stop_f: }
7157 \cs_new_protected:Npn \char_show_value_catcode:n #1
7158 { \exp_args:Nf \tl_show:n { \char_value_catcode:n {#1} } }
```

(End definition for `\char_set_catcode:nn`, `\char_value_catcode:n`, and `\char_show_value_catcode:n`. These functions are documented on page 116.)

```
\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N

7159 \cs_new_protected:Npn \char_set_catcode_escape:N #1
7160 { \char_set_catcode:nn { '#1 } { 0 } }
7161 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
7162 { \char_set_catcode:nn { '#1 } { 1 } }
7163 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
7164 { \char_set_catcode:nn { '#1 } { 2 } }
7165 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
7166 { \char_set_catcode:nn { '#1 } { 3 } }
7167 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
7168 { \char_set_catcode:nn { '#1 } { 4 } }
7169 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
7170 { \char_set_catcode:nn { '#1 } { 5 } }
7171 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
7172 { \char_set_catcode:nn { '#1 } { 6 } }
7173 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
7174 { \char_set_catcode:nn { '#1 } { 7 } }
7175 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
7176 { \char_set_catcode:nn { '#1 } { 8 } }
7177 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
7178 { \char_set_catcode:nn { '#1 } { 9 } }
7179 \cs_new_protected:Npn \char_set_catcode_space:N #1
7180 { \char_set_catcode:nn { '#1 } { 10 } }
7181 \cs_new_protected:Npn \char_set_catcode_letter:N #1
7182 { \char_set_catcode:nn { '#1 } { 11 } }
7183 \cs_new_protected:Npn \char_set_catcode_other:N #1
7184 { \char_set_catcode:nn { '#1 } { 12 } }
7185 \cs_new_protected:Npn \char_set_catcode_active:N #1
7186 { \char_set_catcode:nn { '#1 } { 13 } }
7187 \cs_new_protected:Npn \char_set_catcode_comment:N #1
7188 { \char_set_catcode:nn { '#1 } { 14 } }
7189 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
7190 { \char_set_catcode:nn { '#1 } { 15 } }
```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 115.)

```

\char_set_catcode_escape:n
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n

7191 \cs_new_protected:Npn \char_set_catcode_escape:n #1
7192 { \char_set_catcode:nn {#1} { 0 } }
7193 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
7194 { \char_set_catcode:nn {#1} { 1 } }
7195 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
7196 { \char_set_catcode:nn {#1} { 2 } }
7197 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
7198 { \char_set_catcode:nn {#1} { 3 } }
7199 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
7200 { \char_set_catcode:nn {#1} { 4 } }
7201 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
7202 { \char_set_catcode:nn {#1} { 5 } }
7203 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
7204 { \char_set_catcode:nn {#1} { 6 } }
7205 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
7206 { \char_set_catcode:nn {#1} { 7 } }
7207 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
7208 { \char_set_catcode:nn {#1} { 8 } }
7209 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
7210 { \char_set_catcode:nn {#1} { 9 } }
7211 \cs_new_protected:Npn \char_set_catcode_space:n #1
7212 { \char_set_catcode:nn {#1} { 10 } }
7213 \cs_new_protected:Npn \char_set_catcode_letter:n #1
7214 { \char_set_catcode:nn {#1} { 11 } }
7215 \cs_new_protected:Npn \char_set_catcode_other:n #1
7216 { \char_set_catcode:nn {#1} { 12 } }
7217 \cs_new_protected:Npn \char_set_catcode_active:n #1
7218 { \char_set_catcode:nn {#1} { 13 } }
7219 \cs_new_protected:Npn \char_set_catcode_comment:n #1
7220 { \char_set_catcode:nn {#1} { 14 } }
7221 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
7222 { \char_set_catcode:nn {#1} { 15 } }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 115.)

```

\char_set_mathcode:nn
\char_value_mathcode:n
\char_show_value_mathcode:n
\char_set_lccode:nn
\char_value_lccode:n
\char_show_value_lccode:n
\char_set_uccode:nn
\char_value_uccode:n
\char_show_value_uccode:n
\char_set_sfcode:nn
\char_value_sfcode:n
\char_show_value_sfcode:n

Pretty repetitive, but necessary!

7223 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
7224 { \tex_mathcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
7225 \cs_new:Npn \char_value_mathcode:n #1
7226 { \tex_the:D \tex_mathcode:D \int_eval:n {#1} \exp_stop_f: }
7227 \cs_new_protected:Npn \char_show_value_mathcode:n #1
7228 { \exp_args:Nf \tl_show:n { \char_value_mathcode:n {#1} } }
7229 \cs_new_protected:Npn \char_set_lccode:nn #1#2
7230 { \tex_lccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
7231 \cs_new:Npn \char_value_lccode:n #1
7232 { \tex_the:D \tex_lccode:D \int_eval:n {#1} \exp_stop_f: }
7233 \cs_new_protected:Npn \char_show_value_lccode:n #1
7234 { \exp_args:Nf \tl_show:n { \char_value_lccode:n {#1} } }
7235 \cs_new_protected:Npn \char_set_uccode:nn #1#2
7236 { \tex_uccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
7237 \cs_new:Npn \char_value_uccode:n #1
7238 { \tex_the:D \tex_uccode:D \int_eval:n {#1} \exp_stop_f: }

```

```

7239 \cs_new_protected:Npn \char_show_value_uccode:n #1
7240 { \exp_args:Nf \tl_show:n { \char_value_uccode:n {#1} } }
7241 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
7242 { \tex_sfcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
7243 \cs_new:Npn \char_value_sfcode:n #1
7244 { \tex_the:D \tex_sfcode:D \int_eval:n {#1} \exp_stop_f: }
7245 \cs_new_protected:Npn \char_show_value_sfcode:n #1
7246 { \exp_args:Nf \tl_show:n { \char_value_sfcode:n {#1} } }

```

(End definition for `\char_set_mathcode:nn` and others. These functions are documented on page 117.)

`\l_char_active_seq`
`\l_char_special_seq`

Two sequences for dealing with special characters. The first is characters which may be active, the second longer list is for “special” characters more generally. Both lists are escaped so that for example bulk code assignments can be carried out. In both cases, the order is by ASCII character code (as is done in for example `\ExplSyntaxOn`).

```

7247 \seq_new:N \l_char_special_seq
7248 \seq_set_split:Nnn \l_char_special_seq { }
7249 { \ \ " \# \$ \% \& \ \ ^ \_ \{ \} \~ }
7250 \seq_new:N \l_char_active_seq
7251 \seq_set_split:Nnn \l_char_active_seq { }
7252 { \ " \$ \& \^ \_ \~ }

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 117.)

15.2 Creating character tokens

`\char_set_active_eq:NN`
`\char_set_active_eq:Nc`
`\char_gset_active_eq:NN`
`\char_gset_active_eq:Nc`
`\char_set_active_eq:nN`
`\char_set_active_eq:nc`
`\char_gset_active_eq:nN`
`\char_gset_active_eq:nc`

Four simple functions with very similar definitions, so set up using an auxiliary. These are similar to LuaTeX’s `\lcharcode` primitive.

```

7253 \group_begin:
7254 \char_set_catcode_active:N \^^@
7255 \cs_set_protected:Npn \__char_tmp:nN #1#2
7256 {
7257   \cs_new_protected:cpn { #1 :nN } ##1
7258   {
7259     \group_begin:
7260     \char_set_lccode:nn { \^^@ } { ##1 }
7261     \tex_lowercase:D { \group_end: #2 ^^@ }
7262   }
7263   \cs_new_protected:cpx { #1 :NN } ##1
7264   { \exp_not:c { #1 : nN } { '##1 } }
7265 }
7266 \__char_tmp:nN { char_set_active_eq } \cs_set_eq:NN
7267 \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
7268 \group_end:
7269 \cs_generate_variant:Nn \char_set_active_eq:NN { Nc }
7270 \cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
7271 \cs_generate_variant:Nn \char_set_active_eq:nN { nc }
7272 \cs_generate_variant:Nn \char_gset_active_eq:nN { nc }

```

(End definition for `\char_set_active_eq:NN` and others. These functions are documented on page 113.)

`__char_int_to_roman:w`

For efficiency in 8-bit engines, we use the faster primitive approach to making roman numerals.

```

7273 \cs_new_eq:NN \__char_int_to_roman:w \tex_romannumeral:D

```

(End definition for `_char_int_to_roman:w`.)

`\char_generate:nn` The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (Xe_{La}TeX, Lua_{TeX}). There are though various issues which are covered below. At the interface layer, turn the two arguments into integers up-front so this is only done once.

```

\__char_generate_aux:nn
\__char_generate_aux:nnw
\__char_generate_auxii:nnw
  \l__char_tmp_tl
\__char_generate_invalid_catcode:
7274 \cs_new:Npn \char_generate:nn #1#2
7275 {
7276   \exp:w \exp_after:wN \__char_generate_aux:w
7277   \int_value:w \int_eval:n {#1} \exp_after:wN ;
7278   \int_value:w \int_eval:n {#2} ;
7279 }
```

Before doing any actual conversion, first some special case filtering. Spaces are out here as Lua_{TeX} emulation only makes normal (charcode 32 spaces). However, `^^@` is filtered out separately as that can't be done with macro emulation either, so is flagged up separately. That done, hand off to the engine-dependent part.

```

7280 \cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
7281 {
7282   \if_int_compare:w #2 = 10 \exp_stop_f:
7283     \if_int_compare:w #1 = 0 \exp_stop_f:
7284       \__kernel_msg_expandable_error:nn { kernel } { char-null-space }
7285     \else:
7286       \__kernel_msg_expandable_error:nn { kernel } { char-space }
7287     \fi:
7288   \else:
7289     \if_int_odd:w 0
7290       \if_int_compare:w #2 < 1 \exp_stop_f: 1 \fi:
7291       \if_int_compare:w #2 = 5 \exp_stop_f: 1 \fi:
7292       \if_int_compare:w #2 = 9 \exp_stop_f: 1 \fi:
7293       \if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
7294       \__kernel_msg_expandable_error:nn { kernel }
7295       { char-invalid-catcode }
7296     \else:
7297       \if_int_odd:w 0
7298         \if_int_compare:w #1 < 0 \exp_stop_f: 1 \fi:
7299         \if_int_compare:w #1 > \c_max_char_int 1 \fi: \exp_stop_f:
7300         \__kernel_msg_expandable_error:nn { kernel }
7301         { char-out-of-range }
7302       \else:
7303         \__char_generate_aux:nnw {#1} {#2}
7304       \fi:
7305     \fi:
7306   \fi:
7307   \exp_end:
7308 }
7309 \tl_new:N \l__char_tmp_tl
```

Engine-dependent definitions are now needed for the implementation. For Lua_{TeX} and recent Xe_{La}TeX releases there is engine-level support. They can do cases that macro emulation can't. All of those are filtered out here using a primitive-based boolean expression for speed. The final level is the basic definition at the engine level: the arguments here are integers so there is no need to worry about them too much. At present Xe_{La}TeX cannot

generate active characters so we filter that: at some future stage that may change: the slightly odd ordering of auxiliaries reflects that.

```

7310 \group_begin:
7311 (*package)
7312   \char_set_catcode_active:N \^^L
7313   \cs_set:Npn \^^L { }
7314 \group_end:
7315   \char_set_catcode_other:n { 0 }
7316   \if_int_odd:w 0
7317     \sys_if_engine luatex:T { 1 }
7318     \sys_if_engine xetex:T { 1 } \exp_stop_f:
7319     \sys_if_engine luatex:TF
7320     {
7321       \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
7322       {
7323         #3
7324         \exp_after:wN \exp_after:wN \exp_after:wN \exp_end:
7325         \lua_now_x:n { \kernel.charcat(#1, #2) }
7326       }
7327     }
7328     {
7329       \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
7330       {
7331         #3
7332         \exp_after:wN \exp_end:
7333         \utex_charcat:D #1 \exp_stop_f: #2 \exp_stop_f:
7334       }
7335       \cs_new_eq:NN \__char_generate_auxii:nnw \__char_generate_aux:nnw
7336       \cs_gset:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
7337       {
7338         #3
7339         \if_int_compare:w #2 = 13 \exp_stop_f:
7340           \__kernel_msg_expandable_error:nn { kernel } { char-active }
7341         \else:
7342           \__char_generate_auxii:nnw {#1} {#2}
7343         \fi:
7344         \exp_end:
7345       }
7346     }
7347   \else:

```

For engines where `\Ucharcat` isn't available (or emulated) then we have to work in macros, and cover only the 8-bit range. The first stage is to build up a `tl` containing `^^@` with each category code that can be accessed in this way, with an error set up for the other cases. This is all done such that it can be quickly accessed using a `\if_case:w` low-level conditional. There are a few things to notice here. As `^^L` is `\outer` we need to locally set it to avoid a problem. To get open/close braces into the list, they are set up using `\if_false:` pairing and are then x-type expanded together into the desired form.

```

7348   \tl_set:Nn \l__char_tmp_tl { \exp_not:N \or: }
7349   \char_set_catcode_group_begin:n { 0 } % {
7350   \tl_put_right:Nn \l__char_tmp_tl { ^^@ \if_false: } }
7351   \char_set_catcode_group_end:n { 0 }
7352   \tl_put_right:Nn \l__char_tmp_tl { { \fi: \exp_not:N \or: ^^@ } % }
7353   \tl_set:Nx \l__char_tmp_tl { \l__char_tmp_tl }

```

```

7354 \char_set_catcode_math_toggle:n { 0 }
7355 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }

```

As T_EX is very unhappy if it finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. T_EX is happy if the token is hidden inside `\unexpanded` (which needs to be the primitive). The expansion chain here is required so that the conditional gets cleaned up correctly (other code assumes there is exactly one token to skip during the clean-up).

```

7356 \char_set_catcode_alignment:n { 0 }
7357 \tl_put_right:Nn \l__char_tmp_tl
7358 {
7359   \or:
7360   \__kernel_exp_not:w \exp_after:wN
7361   { \exp_after:wN ^^@ \exp_after:wN }
7362 }
7363 \tl_put_right:Nn \l__char_tmp_tl { \or: }
7364 \char_set_catcode_parameter:n { 0 }
7365 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
7366 \char_set_catcode_math_superscript:n { 0 }
7367 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
7368 \char_set_catcode_math_subscript:n { 0 }
7369 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
7370 \tl_put_right:Nn \l__char_tmp_tl { \or: }

```

For making spaces, there needs to be an o-type expansion of a `\use:n` (or some other tokenization) to avoid dropping the space. We also set up active tokens although they are (currently) filtered out by the interface layer (`\Ucharcat` cannot make active tokens).

```

7371 \char_set_catcode_space:n { 0 }
7372 \tl_put_right:Nn \l__char_tmp_tl { \use:n { \or: } ^^@ }
7373 \char_set_catcode_letter:n { 0 }
7374 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
7375 \char_set_catcode_other:n { 0 }
7376 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
7377 \char_set_catcode_active:n { 0 }
7378 \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }

```

Convert the above temporary list into a series of constant token lists, one for each character code, using `\tex_lowercase:D` to convert `^^@` in each case. The x-type expansion ensures that `\tex_lowercase:D` receives the contents of the token list. In package mode, `^^L` is awkward hence this is done in three parts. Notice that at this stage `^^@` is active.

```

7379 \cs_set_protected:Npn \__char_tmp:n #1
7380 {
7381   \char_set_lccode:nn { 0 } {#1}
7382   \char_set_lccode:nn { 32 } {#1}
7383   \exp_args:Nx \tex_lowercase:D
7384   {
7385     \tl_const:Nn
7386     \exp_not:c { c__char_ \__char_int_to_roman:w #1 _tl }
7387     { \exp_not:o \l__char_tmp_tl }
7388   }
7389 }
7390 (*package)
7391 \int_step_function:nnN { 0 } { 11 } \__char_tmp:n
7392 \group_begin:

```

```

7393     \tl_replace_once:Nnn \l__char_tmp_tl { ^^@ } { \ERROR }
7394     \__char_tmp:n { 12 }
7395     \group_end:
7396     \int_step_function:nnN { 13 } { 255 } \__char_tmp:n
7397 \end{package}
7398 \begin{initex}
7399     \int_step_function:nnN { 0 } { 255 } \__char_tmp:n
7400 \end{initex}
7401     \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
7402     {
7403         #3
7404         \exp_after:wN \exp_after:wN
7405         \exp_after:wN \exp_end:
7406         \exp_after:wN \exp_after:wN
7407         \if_case:w #2
7408             \exp_last_unbraced:Nv \exp_stop_f:
7409             { c__char_ \__char_int_to_roman:w #1 _tl }
7410         \fi:
7411     }
7412     \fi:
7413 \endgroup_end:

```

(End definition for `\char_generate:nn` and others. This function is documented on page 114.)

`\c_catcode_other_space_tl` Create a space with category code 12: an “other” space.

```

7414 \tl_const:Nx \c_catcode_other_space_tl { \char_generate:nn { '\ } { 12 } }

```

(End definition for `\c_catcode_other_space_tl`. This function is documented on page 114.)

15.3 Generic tokens

```

7415 \@@=token

```

`\token_to_meaning:N` These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

```

\token_to_meaning:c
\token_to_str:N
\token_to_str:c

```

(End definition for `\token_to_meaning:N` and `\token_to_str:N`. These functions are documented on page 118.)

`\c_group_begin_token` We define these useful tokens. For the brace and space tokens things have to be done
`\c_group_end_token` by hand: the formal argument spec. for `\cs_new_eq:NN` does not cover them so we do
`\c_math_toggle_token` things by hand. (As currently coded it would *work* with `\cs_new_eq:NN` but that’s not
`\c_alignment_token` really a great idea to show off: we want people to stick to the defined interfaces and that
`\c_parameter_token` includes us.) So that these few odd names go into the log when appropriate there is a
`\c_math_superscript_token` need to hand-apply the `__kernel_chk_if_free_cs:N` check.
`\c_math_subscript_token`
`\c_space_token`
`\c_catcode_letter_token`
`\c_catcode_other_token`

```

7416 \group_begin:
7417     \__kernel_chk_if_free_cs:N \c_group_begin_token
7418     \tex_global:D \tex_let:D \c_group_begin_token {
7419         \__kernel_chk_if_free_cs:N \c_group_end_token
7420         \tex_global:D \tex_let:D \c_group_end_token }
7421     \char_set_catcode_math_toggle:N \*
7422     \cs_new_eq:NN \c_math_toggle_token *
7423     \char_set_catcode_alignment:N \*
7424     \cs_new_eq:NN \c_alignment_token *
7425     \cs_new_eq:NN \c_parameter_token #
7426     \cs_new_eq:NN \c_math_superscript_token ^

```



```

7427 \char_set_catcode_math_subscript:N \*
7428 \cs_new_eq:NN \c_math_subscript_token *
7429 \__kernel_chk_if_free_cs:N \c_space_token
7430 \use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~
7431 \cs_new_eq:NN \c_catcode_letter_token a
7432 \cs_new_eq:NN \c_catcode_other_token 1
7433 \group_end:

```

(End definition for `\c_group_begin_token` and others. These functions are documented on page 118.)

`\c_catcode_active_tl` Not an implicit token!

```

7434 \group_begin:
7435 \char_set_catcode_active:N \*
7436 \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
7437 \group_end:

```

(End definition for `\c_catcode_active_tl`. This variable is documented on page 118.)

15.4 Token conditionals

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.
`\token_if_group_begin:NTF`

```

7438 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
7439 {
7440     \if_catcode:w \exp_not:N #1 \c_group_begin_token
7441     \prg_return_true: \else: \prg_return_false: \fi:
7442 }

```

(End definition for `\token_if_group_begin:NTF`. This function is documented on page 119.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.
`\token_if_group_end:NTF`

```

7443 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
7444 {
7445     \if_catcode:w \exp_not:N #1 \c_group_end_token
7446     \prg_return_true: \else: \prg_return_false: \fi:
7447 }

```

(End definition for `\token_if_group_end:NTF`. This function is documented on page 119.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.
`\token_if_math_toggle:NTF`

```

7448 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
7449 {
7450     \if_catcode:w \exp_not:N #1 \c_math_toggle_token
7451     \prg_return_true: \else: \prg_return_false: \fi:
7452 }

```

(End definition for `\token_if_math_toggle:NTF`. This function is documented on page 119.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.
`\token_if_alignment:NTF`

```

7453 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
7454 {
7455     \if_catcode:w \exp_not:N #1 \c_alignment_token
7456     \prg_return_true: \else: \prg_return_false: \fi:
7457 }

```

(End definition for `\token_if_alignment:NTF`. This function is documented on page 119.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.
`\token_if_parameter:NTF` We have to trick T_EX a bit to avoid an error message: within a group we prevent `\c_parameter_token` from behaving like a macro parameter character. The definitions of `\prg_new_conditional:Npnn` are global, so they remain after the group.

```

7458 \group_begin:
7459 \cs_set_eq:NN \c_parameter_token \scan_stop:
7460 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
7461 {
7462     \if_catcode:w \exp_not:N #1 \c_parameter_token
7463     \prg_return_true: \else: \prg_return_false: \fi:
7464 }
7465 \group_end:

```

(End definition for `\token_if_parameter:NTF`. This function is documented on page 119.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.
`\token_if_math_superscript:NTF`

```

7466 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
7467 { p , T , F , TF }
7468 {
7469     \if_catcode:w \exp_not:N #1 \c_math_superscript_token
7470     \prg_return_true: \else: \prg_return_false: \fi:
7471 }

```

(End definition for `\token_if_math_superscript:NTF`. This function is documented on page 119.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.
`\token_if_math_subscript:NTF`

```

7472 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
7473 {
7474     \if_catcode:w \exp_not:N #1 \c_math_subscript_token
7475     \prg_return_true: \else: \prg_return_false: \fi:
7476 }

```

(End definition for `\token_if_math_subscript:NTF`. This function is documented on page 119.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.

```

\token_if_space:NTF
7477 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
7478 {
7479     \if_catcode:w \exp_not:N #1 \c_space_token
7480     \prg_return_true: \else: \prg_return_false: \fi:
7481 }

```

(End definition for `\token_if_space:NTF`. This function is documented on page 119.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.

```

\token_if_letter:NTF
7482 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
7483 {
7484     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
7485     \prg_return_true: \else: \prg_return_false: \fi:
7486 }

```

(End definition for `\token_if_letter:NTF`. This function is documented on page 120.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token`
`\token_if_other:N \underline{TF}` for this.

```

7487 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
7488 {
7489     \if_catcode:w \exp_not:N #1 \c_catcode_other_token
7490     \prg_return_true: \else: \prg_return_false: \fi:
7491 }

```

(End definition for `\token_if_other:N \underline{TF}` . This function is documented on page 120.)

`\token_if_active_p:N` Check if token is an active char token. We use the constant `\c_catcode_active_tl` for
`\token_if_active:N \underline{TF}` this. A technical point is that `\c_catcode_active_tl` is in fact a macro expanding to
`\exp_not:N *`, where `*` is active.

```

7492 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
7493 {
7494     \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
7495     \prg_return_true: \else: \prg_return_false: \fi:
7496 }

```

(End definition for `\token_if_active:N \underline{TF}` . This function is documented on page 120.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.

`\token_if_eq_meaning:NN \underline{TF}`

```

7497 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
7498 {
7499     \if_meaning:w #1 #2
7500     \prg_return_true: \else: \prg_return_false: \fi:
7501 }

```

(End definition for `\token_if_eq_meaning:NN \underline{TF}` . This function is documented on page 120.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.

`\token_if_eq_catcode:NN \underline{TF}`

```

7502 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
7503 {
7504     \if_catcode:w \exp_not:N #1 \exp_not:N #2
7505     \prg_return_true: \else: \prg_return_false: \fi:
7506 }

```

(End definition for `\token_if_eq_catcode:NN \underline{TF}` . This function is documented on page 120.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.

`\token_if_eq_charcode:NN \underline{TF}`

```

7507 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
7508 {
7509     \if_charcode:w \exp_not:N #1 \exp_not:N #2
7510     \prg_return_true: \else: \prg_return_false: \fi:
7511 }

```

(End definition for `\token_if_eq_charcode:NN \underline{TF}` . This function is documented on page 120.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` always outputs something like
`\token_if_macro:N \underline{TF}` `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`.
`__token_if_macro_p:w` However, this can fail the five `\...mark` primitives, whose meaning has the form
`\...mark:<user material>`. The problem is that the `<user material>` can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have

any combination of `\long`, `\protected` or `\outer` (not used in L^AT_EX3) before the string `macro:`. We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `m...`.

Both `ma` and `:` must be of category code 12 (other), so are detokenized.

```

7512 \use:x
7513 {
7514   \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N #1
7515   { p , T , F , TF }
7516   {
7517     \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
7518     \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma : }
7519     \exp_not:N \q_stop
7520   }
7521   \cs_new:Npn \exp_not:N \__token_if_macro_p:w
7522   ##1 \tl_to_str:n { ma } ##2 \c_colon_str ##3 \exp_not:N \q_stop
7523 }
7524 {
7525   \str_if_eq_x:nnTF { #2 } { cro }
7526   { \prg_return_true: }
7527   { \prg_return_false: }
7528 }

```

(End definition for `\token_if_macro:N` and `__token_if_macro_p:w`. This function is documented on page 120.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as `\token_if_letter:N` etc. We use `\scan_stop:` for this.

`\token_if_cs:N` *TF*

```

7529 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
7530 {
7531   \if_catcode:w \exp_not:N #1 \scan_stop:
7532   \prg_return_true: \else: \prg_return_false: \fi:
7533 }

```

(End definition for `\token_if_cs:N`. This function is documented on page 120.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that T_EX temporarily converts `\exp_not:N` $\langle token \rangle$ into `\scan_stop:` if $\langle token \rangle$ is expandable. An undefined token is not considered as expandable. No problem nesting the conditionals, since the third `#1` is only skipped if it is non-expandable (hence not part of T_EX's conditional apparatus).

`\token_if_expandable:N` *TF*

```

7534 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
7535 {
7536   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
7537   \prg_return_false:
7538   \else:
7539     \if_cs_exist:N #1
7540     \prg_return_true:
7541     \else:
7542     \prg_return_false:
7543     \fi:

```

```

7544     \fi:
7545   }

```

(End definition for `\token_if_expandable:NTF`. This function is documented on page 120.)

`__token_delimit_by_char:w` These auxiliary functions are used below to define some conditionals which detect whether the `\meaning` of their argument begins with a particular string. Each auxiliary takes an argument delimited by a string, a second one delimited by `\q_stop`, and returns the first one and its delimiter. This result is eventually compared to another string.

```

\__token_delimit_by_count:w
\__token_delimit_by_dimen:w
\__token_delimit_by_macro:w
\__token_delimit_by_muskip:w
\__token_delimit_by_skip:w
\__token_delimit_by_toks:w

7546 \group_begin:
7547 \cs_set_protected:Npn \__token_tmp:w #1
7548 {
7549   \use:x
7550   {
7551     \cs_new:Npn \exp_not:c { __token_delimit_by_ #1 :w }
7552     #####1 \tl_to_str:n {#1} #####2 \exp_not:N \q_stop
7553     { #####1 \tl_to_str:n {#1} }
7554   }
7555 }
7556 \__token_tmp:w { char" }
7557 \__token_tmp:w { count }
7558 \__token_tmp:w { dimen }
7559 \__token_tmp:w { macro }
7560 \__token_tmp:w { muskip }
7561 \__token_tmp:w { skip }
7562 \__token_tmp:w { toks }
7563 \group_end:

```

(End definition for `__token_delimit_by_char:w` and others.)

`\token_if_chardef_p:N` Each of these conditionals tests whether its argument's `\meaning` starts with a given string. This is essentially done by having an auxiliary grab an argument delimited by the string and testing whether the argument was empty. Of course, a copy of this string must first be added to the end of the `\meaning` to avoid a runaway argument in case it does not contain the string. Two complications arise. First, the escape character is not fixed, and cannot be included in the delimiter of the auxiliary function (this function cannot be defined on the fly because tests must remain expandable): instead the first argument of the auxiliary (plus the delimiter to avoid complications with trailing spaces) is compared using `\str_if_eq:x:nnTF` to the result of applying `\token_to_str:N` to a control sequence. Second, the `\meaning` of primitives such as `\dimen` or `\dimendef` starts in the same way as registers such as `\dimen123`, so they must be tested for.

Characters used as delimiters must have catcode 12 and are obtained through `\tl_to_str:n`. This requires doing all definitions within `x`-expansion. The temporary function `__token_tmp:w` used to define each conditional receives three arguments: the name of the conditional, the auxiliary's delimiter (also used to name the auxiliary), and the string to which one compares the auxiliary's result. Note that the `\meaning` of a protected long macro starts with `\protected\long macro`, with no space after `\protected` but a space after `\long`, hence the mixture of `\token_to_str:N` and `\tl_to_str:n`.

For the first five conditionals, `\cs_if_exist:cT` turns out to be `false`, and the code boils down to a string comparison between the result of the auxiliary on the `\meaning` of the conditional's argument `#####1`, and `#3`. Both are evaluated at run-time, as this is important to get the correct escape character.

The other five conditionals have additional code that compares the argument #####1 to two T_EX primitives which would wrongly be recognized as registers otherwise. Despite using T_EX's primitive conditional construction, this does not break when #####1 is itself a conditional, because branches of the conditionals are only skipped if #####1 is one of the two primitives that are tested for (which are not T_EX conditionals).

```

7564 \group_begin:
7565 \cs_set_protected:Npn \__token_tmp:w #1#2#3
7566 {
7567   \use:x
7568   {
7569     \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } #####1
7570     { p , T , F , TF }
7571     {
7572       \cs_if_exist:cT { tex_ #2 :D }
7573       {
7574         \exp_not:N \if_meaning:w #####1 \exp_not:c { tex_ #2 :D }
7575         \exp_not:N \prg_return_false:
7576         \exp_not:N \else:
7577         \exp_not:N \if_meaning:w #####1 \exp_not:c { tex_ #2 def:D }
7578         \exp_not:N \prg_return_false:
7579         \exp_not:N \else:
7580         }
7581         \exp_not:N \str_if_eq_x:nnTF
7582         {
7583           \exp_not:N \exp_after:wN
7584           \exp_not:c { __token_delimit_by_ #2 :w }
7585           \exp_not:N \token_to_meaning:N #####1
7586           ? \tl_to_str:n {#2} \exp_not:N \q_stop
7587         }
7588         { \exp_not:n {#3} }
7589         { \exp_not:N \prg_return_true: }
7590         { \exp_not:N \prg_return_false: }
7591         \cs_if_exist:cT { tex_ #2 :D }
7592         {
7593           \exp_not:N \fi:
7594           \exp_not:N \fi:
7595         }
7596       }
7597     }
7598   }
7599   __token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
7600   __token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
7601   __token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
7602   __token_tmp:w { protected_macro } { macro }
7603   { \tl_to_str:n { \protected } macro }
7604   __token_tmp:w { protected_long_macro } { macro }
7605   { \token_to_str:N \protected \tl_to_str:n { \long } macro }
7606   __token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
7607   __token_tmp:w { int_register } { count } { \token_to_str:N \count }
7608   __token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
7609   __token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
7610   __token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
7611 \group_end:

```

(End definition for `\token_if_chardef:NTF` and others. These functions are documented on page 121.)

`\token_if_primitive_p:N`

`\token_if_primitive:NTF`

`__token_if_primitive:NNw`

`__token_if_primitive_space:w`

`__token_if_primitive_nullfont:N`

`__token_if_primitive_loop:N`

`__token_if_primitive:Nw`

`__token_if_primitive_undefined:N`

We filter out macros first, because they cause endless trouble later otherwise.

Primitives are almost distinguished by the fact that the result of `\token_to_meaning:N` is formed from letters only. Every other token has either a space (e.g., the letter A), a digit (e.g., `\count123`) or a double quote (e.g., `\char"A`).

Ten exceptions: on the one hand, `\tex_undefined:D` is not a primitive, but its meaning is undefined, only letters; on the other hand, `\space`, `\italiccorr`, `\hyphen`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark`, `\splitbotmark`, and `\nullfont` are primitives, but have non-letters in their meaning.

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be nonexistent depending on `\endlinechar`), and takes care of three of the exceptions: `\space`, `\italiccorr` and `\hyphen`, whose meaning is at most two characters. This leaves a string terminated by some `:`, and `\q_stop`.

The meaning of each one of the five `\...mark` primitives has the form $\langle letters \rangle : \langle user material \rangle$. In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either `"`, or a space, or a digit. Two exceptions remain: `\tex_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than ‘A’ (this is not quite a test for “only letters”, but is close enough to work in this context). If this first character is `:` then we have a primitive, or `\tex_undefined:D`, and if it is `"` or a digit, then the token is not a primitive.

```

7612 \tex_chardef:D \c__token_A_int = 'A ~ %
7613 \use:x
7614 {
7615   \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N ##1
7616   { p , T , F , TF }
7617   {
7618     \exp_not:N \token_if_macro:NTF ##1
7619     \exp_not:N \prg_return_false:
7620     {
7621       \exp_not:N \exp_after:wN \exp_not:N \__token_if_primitive:NNw
7622       \exp_not:N \token_to_meaning:N ##1
7623       \tl_to_str:n { : : : } \exp_not:N \q_stop ##1
7624     }
7625   }
7626   \cs_new:Npn \exp_not:N \__token_if_primitive:NNw
7627   ##1##2 ##3 \c_colon_str ##4 \exp_not:N \q_stop
7628   {
7629     \exp_not:N \tl_if_empty:oTF
7630     { \exp_not:N \__token_if_primitive_space:w ##3 ~ }
7631     {
7632       \exp_not:N \__token_if_primitive_loop:N ##3
7633       \c_colon_str \exp_not:N \q_stop
7634     }
7635     { \exp_not:N \__token_if_primitive_nullfont:N }
7636   }

```

```

7637 }
7638 \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
7639 \cs_new:Npn \__token_if_primitive_nullfont:N #1
7640 {
7641   \if_meaning:w \tex_nullfont:D #1
7642     \prg_return_true:
7643   \else:
7644     \prg_return_false:
7645   \fi:
7646 }
7647 \cs_new:Npn \__token_if_primitive_loop:N #1
7648 {
7649   \if_int_compare:w '#1 < \c__token_A_int %
7650     \exp_after:wN \__token_if_primitive:Nw
7651     \exp_after:wN #1
7652   \else:
7653     \exp_after:wN \__token_if_primitive_loop:N
7654   \fi:
7655 }
7656 \cs_new:Npn \__token_if_primitive:Nw #1 #2 \q_stop
7657 {
7658   \if:w : #1
7659     \exp_after:wN \__token_if_primitive_undefined:N
7660   \else:
7661     \prg_return_false:
7662     \exp_after:wN \use_none:n
7663   \fi:
7664 }
7665 \cs_new:Npn \__token_if_primitive_undefined:N #1
7666 {
7667   \if_cs_exist:N #1
7668     \prg_return_true:
7669   \else:
7670     \prg_return_false:
7671   \fi:
7672 }

```

(End definition for `\token_if_primitive:NTF` and others. This function is documented on page [122](#).)

15.5 Peeking ahead at the next token

7673 `<@@=peek>`

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

\l_peek_token Storage tokens which are publicly documented: the token peeked.

\g_peek_token 7674 \cs_new_eq:NN \l_peek_token ?
7675 \cs_new_eq:NN \g_peek_token ?

(End definition for \l_peek_token and \g_peek_token. These variables are documented on page 122.)

\l__peek_search_token The token to search for as an implicit token: cf. **\l__peek_search_tl**.

7676 \cs_new_eq:NN \l__peek_search_token ?

(End definition for \l__peek_search_token.)

\l__peek_search_tl The token to search for as an explicit token: cf. **\l__peek_search_token**.

7677 \tl_new:N \l__peek_search_tl

(End definition for \l__peek_search_tl.)

__peek_true:w Functions used by the branching and space-stripping code.

__peek_true_aux:w 7678 \cs_new:Npn __peek_true:w { }
__peek_false:w 7679 \cs_new:Npn __peek_true_aux:w { }
__peek_tmp:w 7680 \cs_new:Npn __peek_false:w { }
7681 \cs_new:Npn __peek_tmp:w { }

(End definition for __peek_true:w and others.)

\peek_after:Nw Simple wrappers for **\futurelet**: no arguments absorbed here.

\peek_gafter:Nw 7682 \cs_new_protected:Npn \peek_after:Nw
7683 { \tex_futurelet:D \l_peek_token }
7684 \cs_new_protected:Npn \peek_gafter:Nw
7685 { \tex_global:D \tex_futurelet:D \g_peek_token }

(End definition for \peek_after:Nw and \peek_gafter:Nw. These functions are documented on page 122.)

__peek_true_remove:w A function to remove the next token and then regain control.

7686 \cs_new_protected:Npn __peek_true_remove:w
7687 {
7688 \tex_afterassignment:D __peek_true_aux:w
7689 \cs_set_eq:NN __peek_tmp:w
7690 }

(End definition for __peek_true_remove:w.)

__peek_token_generic_aux:NNNTF The generic functions store the test token in both implicit and explicit modes, and the **true** and **false** code as token lists, more or less. The two branches have to be absorbed here as the input stream needs to be cleared for the peek function itself. Here, #1 is **__peek_true_remove:w** when removing the token and **__peek_true_aux:w** otherwise.

7691 \cs_new_protected:Npn __peek_token_generic_aux:NNNTF #1#2#3#4#5
7692 {
7693 \group_align_safe_begin:
7694 \cs_set_eq:NN \l__peek_search_token #3
7695 \tl_set:Nn \l__peek_search_tl {#3}
7696 \cs_set:Npx __peek_true_aux:w
7697 {
7698 \exp_not:N \group_align_safe_end:
7699 \exp_not:n {#4}

```

7700     }
7701     \cs_set_eq:NN \__peek_true:w #1
7702     \cs_set:Npx \__peek_false:w
7703     {
7704         \exp_not:N \group_align_safe_end:
7705         \exp_not:n {#5}
7706     }
7707     \peek_after:Nw #2
7708 }

```

(End definition for __peek_token_generic_aux:NNTF.)

__peek_token_generic:NNTF For token removal there needs to be a call to the auxiliary function which does the work.

```

\__peek_token_remove_generic:NNTF
7709 \cs_new_protected:Npn \__peek_token_generic:NNTF
7710 { \__peek_token_generic_aux:NNTF \__peek_true_aux:w }
7711 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
7712 { \__peek_token_generic:NNTF #1 #2 {#3} { } }
7713 \cs_new_protected:Npn \__peek_token_generic:NNF #1#2#3
7714 { \__peek_token_generic:NNTF #1 #2 { } {#3} }
7715 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF
7716 { \__peek_token_generic_aux:NNTF \__peek_true_remove:w }
7717 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
7718 { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
7719 \cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
7720 { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }

```

(End definition for __peek_token_generic:NNTF and __peek_token_remove_generic:NNTF.)

__peek_execute_branches_meaning: The meaning test is straight forward.

```

7721 \cs_new:Npn \__peek_execute_branches_meaning:
7722 {
7723     \if_meaning:w \l_peek_token \l_peek_search_token
7724     \exp_after:wN \__peek_true:w
7725     \else:
7726     \exp_after:wN \__peek_false:w
7727     \fi:
7728 }

```

(End definition for __peek_execute_branches_meaning:.)

__peek_execute_branches_catcode: The catcode and charcode tests are very similar, and in order to use the same auxiliaries we do something a little bit odd, firing \if_catcode:w and \if_charcode:w before finding the operands for those tests, which are only given in the auxii:N and auxiii: auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:

- control sequences which are not equal to a non-active character token (*e.g.*, macro, primitive);
- active characters which are not equal to a non-active character token (*e.g.*, macro, primitive);
- explicit non-active character tokens, or control sequences or active characters set equal to a non-active character token.

The first two cases are not distinguishable simply using TeX's `\futurelet`, because we can only access the `\meaning` of tokens in that way. In those cases, detected thanks to a comparison with `\scan_stop:`, we grab the following token, and compare it explicitly with the explicit search token stored in `\l_peek_search_tl`. The `\exp_not:N` prevents outer macros (coming from non-L^AT_EX3 code) from blowing up. In the third case, `\l_peek_token` is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```

7729 \cs_new:Npn \__peek_execute_branches_catcode:
7730 { \if_catcode:w \__peek_execute_branches_catcode_aux: }
7731 \cs_new:Npn \__peek_execute_branches_charcode:
7732 { \if_charcode:w \__peek_execute_branches_catcode_aux: }
7733 \cs_new:Npn \__peek_execute_branches_catcode_aux:
7734 {
7735     \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
7736     \exp_after:wN \exp_after:wN
7737     \exp_after:wN \__peek_execute_branches_catcode_auxii:N
7738     \exp_after:wN \exp_not:N
7739     \else:
7740     \exp_after:wN \__peek_execute_branches_catcode_auxiii:
7741     \fi:
7742 }
7743 \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
7744 {
7745     \exp_not:N #1
7746     \exp_after:wN \exp_not:N \l_peek_search_tl
7747     \exp_after:wN \__peek_true:w
7748     \else:
7749     \exp_after:wN \__peek_false:w
7750     \fi:
7751     #1
7752 }
7753 \cs_new:Npn \__peek_execute_branches_catcode_auxiii:
7754 {
7755     \exp_not:N \l_peek_token
7756     \exp_after:wN \exp_not:N \l_peek_search_tl
7757     \exp_after:wN \__peek_true:w
7758     \else:
7759     \exp_after:wN \__peek_false:w
7760     \fi:
7761 }

```

(End definition for `__peek_execute_branches_catcode:` and others.)

`__peek_ignore_spaces_execute_branches:` This function removes one space token at a time, and calls `__peek_execute_branches:` when encountering the first non-space token. We directly use the primitive meaning test rather than `\token_if_eq_meaning:NNTF` because `\l_peek_token` may be an outer macro (coming from non-L^AT_EX3 packages). Spaces are removed using a side-effect of f-expansion: `\exp:w \exp_end_continue_f:w` removes one space.

```

7762 \cs_new_protected:Npn \__peek_ignore_spaces_execute_branches:
7763 {
7764     \if_meaning:w \l_peek_token \c_space_token
7765     \exp_after:wN \peek_after:Nw

```

```

7766     \exp_after:wN \__peek_ignore_spaces_execute_branches:
7767     \exp:w \exp_end_continue_f:w
7768     \else:
7769         \exp_after:wN \__peek_execute_branches:
7770     \fi:
7771 }

```

(End definition for __peek_ignore_spaces_execute_branches:.)

__peek_def:nnnn The public functions themselves cannot be defined using \prg_new_conditional:Npnn and so a couple of auxiliary functions are used. As a result, everything is done inside a group. As a result things are a bit complicated.

```

7772 \group_begin:
7773     \cs_set:Npn \__peek_def:nnnn #1#2#3#4
7774     {
7775         \__peek_def:nnnnn {#1} {#2} {#3} {#4} { TF }
7776         \__peek_def:nnnnn {#1} {#2} {#3} {#4} { T }
7777         \__peek_def:nnnnn {#1} {#2} {#3} {#4} { F }
7778     }
7779     \cs_set:Npn \__peek_def:nnnnn #1#2#3#4#5
7780     {
7781         \cs_new_protected:cpx { #1 #5 }
7782         {
7783             \tl_if_empty:nF {#2}
7784             { \exp_not:n { \cs_set_eq:NN \__peek_execute_branches: #2 } }
7785             \exp_not:c { #3 #5 }
7786             \exp_not:n {#4}
7787         }
7788     }

```

(End definition for __peek_def:nnnn and __peek_def:nnnnn.)

\peek_catcode:N~~TF~~ With everything in place the definitions can take place. First for category codes.
\peek_catcode_ignore_spaces:N~~TF~~
\peek_catcode_remove:N~~TF~~
\peek_catcode_remove_ignore_spaces:N~~TF~~

```

7789     \__peek_def:nnnn { peek_catcode:N }
7790     { }
7791     { __peek_token_generic:NN }
7792     { \__peek_execute_branches_catcode: }
7793     \__peek_def:nnnn { peek_catcode_ignore_spaces:N }
7794     { \__peek_execute_branches_catcode: }
7795     { __peek_token_generic:NN }
7796     { \__peek_ignore_spaces_execute_branches: }
7797     \__peek_def:nnnn { peek_catcode_remove:N }
7798     { }
7799     { __peek_token_remove_generic:NN }
7800     { \__peek_execute_branches_catcode: }
7801     \__peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }
7802     { \__peek_execute_branches_catcode: }
7803     { __peek_token_remove_generic:NN }
7804     { \__peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_catcode:N~~TF~~ and others. These functions are documented on page 122.)

\peek_charcode:N~~TF~~ Then for character codes.
\peek_charcode_ignore_spaces:N~~TF~~
\peek_charcode_remove:N~~TF~~
\peek_charcode_remove_ignore_spaces:N~~TF~~

```

7805     \__peek_def:nnnn { peek_charcode:N }
7806     { }

```

```

7807     { __peek_token_generic:NN }
7808     { \__peek_execute_branches_charcode: }
7809     \__peek_def:nnnn { peek_charcode_ignore_spaces:N }
7810     { \__peek_execute_branches_charcode: }
7811     { __peek_token_generic:NN }
7812     { \__peek_ignore_spaces_execute_branches: }
7813     \__peek_def:nnnn { peek_charcode_remove:N }
7814     { }
7815     { __peek_token_remove_generic:NN }
7816     { \__peek_execute_branches_charcode: }
7817     \__peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
7818     { \__peek_execute_branches_charcode: }
7819     { __peek_token_remove_generic:NN }
7820     { \__peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_charcode:NTF and others. These functions are documented on page 123.)

```

\peek_meaning:NTF Finally for meaning, with the group closed to remove the temporary definition functions.
\peek_meaning_ignore_spaces:NTF
\peek_meaning_remove:NTF
\peek_meaning_remove_ignore_spaces:NTF
7821     \__peek_def:nnnn { peek_meaning:N }
7822     { }
7823     { __peek_token_generic:NN }
7824     { \__peek_execute_branches_meaning: }
7825     \__peek_def:nnnn { peek_meaning_ignore_spaces:N }
7826     { \__peek_execute_branches_meaning: }
7827     { __peek_token_generic:NN }
7828     { \__peek_ignore_spaces_execute_branches: }
7829     \__peek_def:nnnn { peek_meaning_remove:N }
7830     { }
7831     { __peek_token_remove_generic:NN }
7832     { \__peek_execute_branches_meaning: }
7833     \__peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
7834     { \__peek_execute_branches_meaning: }
7835     { __peek_token_remove_generic:NN }
7836     { \__peek_ignore_spaces_execute_branches: }
7837     \group_end:

```

(End definition for \peek_meaning:NTF and others. These functions are documented on page 124.)

15.6 Decomposing a macro definition

\token_get_prefix_spec:N We sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the argument specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token \scan_stop: is returned instead.

```

7838 \exp_args:Nno \use:nn
7839 { \cs_new:Npn \__peek_get_prefix_arg_replacement:wN #1 }
7840 { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
7841 { #4 {#1} {#2} {#3} }
7842 \cs_new:Npn \token_get_prefix_spec:N #1
7843 {
7844     \token_if_macro:NTF #1
7845     {

```

```

7846         \exp_after:wN \__peek_get_prefix_arg_replacement:wN
7847         \token_to_meaning:N #1 \q_stop \use_i:nnn
7848     }
7849     { \scan_stop: }
7850 }
7851 \cs_new:Npn \token_get_arg_spec:N #1
7852 {
7853     \token_if_macro:NTF #1
7854     {
7855         \exp_after:wN \__peek_get_prefix_arg_replacement:wN
7856         \token_to_meaning:N #1 \q_stop \use_ii:nnn
7857     }
7858     { \scan_stop: }
7859 }
7860 \cs_new:Npn \token_get_replacement_spec:N #1
7861 {
7862     \token_if_macro:NTF #1
7863     {
7864         \exp_after:wN \__peek_get_prefix_arg_replacement:wN
7865         \token_to_meaning:N #1 \q_stop \use_iii:nnn
7866     }
7867     { \scan_stop: }
7868 }

```

(End definition for `\token_get_prefix_spec:N` and others. These functions are documented on page 125.)

15.7 Deprecated functions

`\token_new:Nn` For removal after 2018-12-31.

```

7869 \__kernel_patch_deprecation:nnNNpn { 2018-12-31 } { \cs_new_eq:NN }
7870 \cs_new_protected:Npn \token_new:Nn #1#2 { \cs_new_eq:NN #1 #2 }

```

(End definition for `\token_new:Nn`.)

```

7871 </initex | package>

```

16 l3prop implementation

The following test files are used for this code: `m3prop001`, `m3prop002`, `m3prop003`, `m3prop004`, `m3show001`.

```

7872 <*initex | package>
7873 <@@=prop>

```

A property list is a macro whose top-level expansion is of the form

```

\__prop \__prop_pair:wn <key1> \s__prop {<value1>}
...
\__prop_pair:wn <keyn> \s__prop {<valuen>}

```

where `\s__prop` is a scan mark (equal to `\scan_stop:`), and `__prop_pair:wn` can be used to map through the property list.

<code>\s__prop</code>	<p>The internal token used at the beginning of property lists. This is also used after each <code><key></code> (see <code>__prop_pair:wn</code>).</p> <p>(End definition for <code>\s__prop</code>.)</p>
<code>__prop_pair:wn</code>	<p><code>__prop_pair:wn <key> \s__prop {<item>}</code></p> <p>The internal token used to begin each key–value pair in the property list. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.</p> <p>(End definition for <code>__prop_pair:wn</code>.)</p>
<code>\l__prop_internal_tl</code>	<p>Token list used to store new key–value pairs to be inserted by functions of the <code>\prop_put:Nnn</code> family.</p> <p>(End definition for <code>\l__prop_internal_tl</code>.)</p>
<hr/> <code>__prop_split:NnTF</code> <hr/>	<code>__prop_split:NnTF <property list> {<key>} {<true code>} {<false code>}</code>
<hr/> Updated: 2013-01-08 <hr/>	<p>Splits the <code><property list></code> at the <code><key></code>, giving three token lists: the <code><extract></code> of <code><property list></code> before the <code><key></code>, the <code><value></code> associated with the <code><key></code> and the <code><extract></code> of the <code><property list></code> after the <code><value></code>. Both <code><extracts></code> retain the internal structure of a property list, and the concatenation of the two <code><extracts></code> is a property list. If the <code><key></code> is present in the <code><property list></code> then the <code><true code></code> is left in the input stream, with <code>#1</code>, <code>#2</code>, and <code>#3</code> replaced by the first <code><extract></code>, the <code><value></code>, and the second <code><extract></code>. If the <code><key></code> is not present in the <code><property list></code> then the <code><false code></code> is left in the input stream, with no trailing material. Both <code><true code></code> and <code><false code></code> are used in the replacement text of a macro defined internally, hence macro parameter characters should be doubled, except <code>#1</code>, <code>#2</code>, and <code>#3</code> which stand in the <code><true code></code> for the three extracts from the property list. The <code><key></code> comparison takes place as described for <code>\str_if_eq:nn</code>.</p>
<code>\s__prop</code>	<p>A private scan mark is used as a marker after each key, and at the very beginning of the property list.</p> <p><small>7874</small> <code>\scan_new:N \s__prop</code></p> <p>(End definition for <code>\s__prop</code>.)</p>
<code>__prop_pair:wn</code>	<p>The delimiter is always defined, but when misused simply triggers an error and removes its argument.</p> <p><small>7875</small> <code>\cs_new:Npn __prop_pair:wn #1 \s__prop #2</code> <small>7876</small> <code>{ __kernel_msg_expandable_error:nn { kernel } { misused-prop } }</code></p> <p>(End definition for <code>__prop_pair:wn</code>.)</p>
<code>\l__prop_internal_tl</code>	<p>Token list used to store the new key–value pair inserted by <code>\prop_put:Nnn</code> and friends.</p> <p><small>7877</small> <code>\tl_new:N \l__prop_internal_tl</code></p> <p>(End definition for <code>\l__prop_internal_tl</code>.)</p>
<code>\c_empty_prop</code>	<p>An empty prop.</p> <p><small>7878</small> <code>\tl_const:Nn \c_empty_prop { \s__prop }</code></p> <p>(End definition for <code>\c_empty_prop</code>. This variable is documented on page 133.)</p>

16.1 Allocation and initialisation

\prop_new:N Property lists are initialized with the value `\c_empty_prop`.

```
\prop_new:c      7879 \cs_new_protected:Npn \prop_new:N #1
                  7880 {
                  7881     \__kernel_chk_if_free_cs:N #1
                  7882     \cs_gset_eq:NN #1 \c_empty_prop
                  7883 }
                  7884 \cs_generate_variant:Nn \prop_new:N { c }
```

(End definition for `\prop_new:N`. This function is documented on page 128.)

\prop_clear:N The same idea for clearing.

```
\prop_clear:c    7885 \cs_new_protected:Npn \prop_clear:N #1
\prop_gclear:N    7886 { \prop_set_eq:NN #1 \c_empty_prop }
\prop_gclear:c    7887 \cs_generate_variant:Nn \prop_clear:N { c }
                  7888 \cs_new_protected:Npn \prop_gclear:N #1
                  7889 { \prop_gset_eq:NN #1 \c_empty_prop }
                  7890 \cs_generate_variant:Nn \prop_gclear:N { c }
```

(End definition for `\prop_clear:N` and `\prop_gclear:N`. These functions are documented on page 128.)

\prop_clear_new:N Once again a simple variation of the token list functions.

```
\prop_clear_new:c 7891 \cs_new_protected:Npn \prop_clear_new:N #1
\prop_gclear_new:N 7892 { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\prop_gclear_new:c 7893 \cs_generate_variant:Nn \prop_clear_new:N { c }
                  7894 \cs_new_protected:Npn \prop_gclear_new:N #1
                  7895 { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
                  7896 \cs_generate_variant:Nn \prop_gclear_new:N { c }
```

(End definition for `\prop_clear_new:N` and `\prop_gclear_new:N`. These functions are documented on page 128.)

\prop_set_eq:NN These are simply copies from the token list functions.

```
\prop_set_eq:cN    7897 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
\prop_set_eq:Nc     7898 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
\prop_set_eq:cc     7899 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
\prop_gset_eq:NN    7900 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
\prop_gset_eq:cN    7901 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\prop_gset_eq:Nc    7902 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\prop_gset_eq:cN    7903 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
\prop_gset_eq:cc    7904 \cs_new_eq:NN \prop_gset_eq:cc \tl_gset_eq:cc
```

(End definition for `\prop_set_eq:NN` and `\prop_gset_eq:NN`. These functions are documented on page 128.)

\l_tmpa_prop We can now initialize the scratch variables.

```
\l_tmpb_prop      7905 \prop_new:N \l_tmpa_prop
\g_tmpa_prop       7906 \prop_new:N \l_tmpb_prop
\g_tmpb_prop       7907 \prop_new:N \g_tmpa_prop
                  7908 \prop_new:N \g_tmpb_prop
```

(End definition for `\l_tmpa_prop` and others. These variables are documented on page 133.)


```

\prop_set_from_keyval:Nn Loop through items separated by commas, with \q_mark to avoid losing braces. After
\prop_set_from_keyval:cn checking for termination, split the item at the first then at the second = (which ought
\prop_gset_from_keyval:Nn to be the first of the trailing =). At both splits, trim spaces and call \__prop_from_
\prop_gset_from_keyval:cn keyval_key:w, then \__prop_from_keyval_value:w, followed by the trimmed material,
\prop_const_from_keyval:Nn \q_nil, the subsequent part of the item, and the trailing ='s and \q_stop. After finding
\prop_const_from_keyval:cn the <key> just store it after \q_stop. After finding the <value> ignore completely empty
  \__prop_from_keyval:n items (both trailing = were used as delimiters and all parts are empty); if the remaining
  \__prop_from_keyval_loop:w part #2 consists exactly of the second trailing = (namely there was exactly one = in the
\__prop_from_keyval_split:Nw item) then output one key-value pair for the property list; otherwise complain about a
  \__prop_from_keyval_key:n missing or extra =.
  \__prop_from_keyval_key:w
  \__prop_from_keyval_value:n
  \__prop_from_keyval_value:w
7909 \cs_new_protected:Npn \prop_set_from_keyval:Nn #1#2
7910 { \tl_set:Nx #1 { \__prop_from_keyval:n {#2} } }
7911 \cs_generate_variant:Nn \prop_set_from_keyval:Nn { c }
7912 \cs_new_protected:Npn \prop_gset_from_keyval:Nn #1#2
7913 { \tl_gset:Nx #1 { \__prop_from_keyval:n {#2} } }
7914 \cs_generate_variant:Nn \prop_gset_from_keyval:Nn { c }
7915 \cs_new_protected:Npn \prop_const_from_keyval:Nn #1#2
7916 { \tl_const:Nx #1 { \__prop_from_keyval:n {#2} } }
7917 \cs_generate_variant:Nn \prop_const_from_keyval:Nn { c }
7918 \cs_new:Npn \__prop_from_keyval:n #1
7919 {
7920   \s__prop
7921   \__prop_from_keyval_loop:w \q_mark #1 ,
7922   \q_recursion_tail , \q_recursion_stop
7923 }
7924 \cs_new:Npn \__prop_from_keyval_loop:w #1 ,
7925 {
7926   \quark_if_recursion_tail_stop:o { \use_none:n #1 }
7927   \__prop_from_keyval_split:Nw \__prop_from_keyval_key:n
7928   #1 = = \q_stop { \use_none:n #1 }
7929   \__prop_from_keyval_loop:w \q_mark
7930 }
7931 \cs_new:Npn \__prop_from_keyval_split:Nw #1#2 =
7932 {
7933   \tl_trim_spaces_apply:oN { \use_none:n #2 } #1
7934   \q_nil
7935 }
7936 \cs_new:Npn \__prop_from_keyval_key:n #1
7937 { \__prop_from_keyval_key:w #1 }
7938 \cs_new:Npn \__prop_from_keyval_key:w #1 \q_nil #2 \q_stop
7939 {
7940   \__prop_from_keyval_split:Nw \__prop_from_keyval_value:n
7941   \q_mark #2 \q_stop {#1}
7942 }
7943 \cs_new:Npn \__prop_from_keyval_value:n #1
7944 { \__prop_from_keyval_value:w #1 }
7945 \cs_new:Npn \__prop_from_keyval_value:w #1 \q_nil #2 \q_stop #3#4
7946 {
7947   \tl_if_empty:nF { #3 #1 #2 }
7948   {
7949     \str_if_eq:nnTF {#2} { = }
7950     {
7951       \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}

```

```

7952         \s__prop { \exp_not:n {#1} }
7953     }
7954 {
7955     \__kernel_msg_expandable_error:nmf
7956     { kernel } { prop-keyval }
7957     { \exp_after:wN \exp_stop_f: #4 }
7958 }
7959 }
7960 }

```

(End definition for `\prop_set_from_keyval:Nn` and others. These functions are documented on page 240.)

16.2 Accessing data in property lists

<pre> __prop_split:NnTF __prop_split_aux:NnTF __prop_split_aux:w </pre>	<p>This function is used by most of the module, and hence must be fast. It receives a $\langle property\ list \rangle$, a $\langle key \rangle$, a $\langle true\ code \rangle$ and a $\langle false\ code \rangle$. The aim is to split the $\langle property\ list \rangle$ at the given $\langle key \rangle$ into the $\langle extract_1 \rangle$ before the key–value pair, the $\langle value \rangle$ associated with the $\langle key \rangle$ and the $\langle extract_2 \rangle$ after the key–value pair. This is done using a delimited function, whose definition is as follows, where the $\langle key \rangle$ is turned into a string.</p>
--	---

```
\cs_set:Npn \__prop_split_aux:w #1
\__prop_pair:wn <key> \s__prop #2
#3 \q_mark #4 #5 \q_stop
{ #4 {\true code} {\false code} }
```

If the $\langle key \rangle$ is present in the property list, `_prop_split_aux:w's #1` is the part before the $\langle key \rangle$, `#2` is the $\langle value \rangle$, `#3` is the part after the $\langle key \rangle$, `#4` is `\use_i:nn`, and `#5` is additional tokens that we do not care about. The $\langle true\ code \rangle$ is left in the input stream, and can use the parameters `#1`, `#2`, `#3` for the three parts of the property list as desired. Namely, the original property list is in this case `#1 _prop_pair:wn $\langle key \rangle$ _s_prop {#2} #3`.

If the $\langle key \rangle$ is not there, then the $\langle function \rangle$ is `\use_ii:nn`, which keeps the $\langle false code \rangle$.

```

7961 \cs_new_protected:Npn \__prop_split:NnTF #1#2
7962   { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
7963 \cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
7964   {
7965     \cs_set:Npn \__prop_split_aux:w ##1
7966       \__prop_pair:wn #2 \s__prop ##2 ##3 \q_mark ##4 ##5 \q_stop
7967       { ##4 {#3} {#4} }
7968     \exp_after:wN \__prop_split_aux:w #1 \q_mark \use_i:nn
7969     \__prop_pair:wn #2 \s__prop { } \q_mark \use_ii:nn \q_stop
7970   }
7971 \cs_new:Npn \__prop_split_aux:w { }

```

(End definition for `__prop_split:NnTF`, `__prop_split_aux:NnTF`, and `__prop_split_aux:w`.)

<pre> \prop_remove:Nn \prop_remove:NV \prop_remove:cn \prop_remove:cV \prop_gremove:Nn \prop_gremove:NV \prop_gremove:cn \prop_gremove:cV </pre>	<p>Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.</p> <pre> 7972 \cs_new_protected:Npn \prop_remove:Nn #1#2 7973 { 7974 __prop_split:NnTF #1 {#2} </pre>
--	--

490

```

7975         { \tl_set:Nn #1 { ##1 ##3 } }
7976     { }
7977 }
7978 \cs_new_protected:Npn \prop_gremove:Nn #1#2
7979 {
7980     \__prop_split:NnTF #1 {#2}
7981     { \tl_gset:Nn #1 { ##1 ##3 } }
7982     { }
7983 }
7984 \cs_generate_variant:Nn \prop_remove:Nn { NV }
7985 \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
7986 \cs_generate_variant:Nn \prop_gremove:Nn { NV }
7987 \cs_generate_variant:Nn \prop_gremove:Nn { c , cV }

```

(End definition for `\prop_remove:Nn` and `\prop_gremove:Nn`. These functions are documented on page 130.)

`\prop_get:NnN` Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to `\q_no_value`.

```

\prop_get:NVN
\prop_get:NoN
\prop_get:cnN
\prop_get:cVN
\prop_get:coN
7988 \cs_new_protected:Npn \prop_get:NnN #1#2#3
7989 {
7990     \__prop_split:NnTF #1 {#2}
7991     { \tl_set:Nn #3 {##2} }
7992     { \tl_set:Nn #3 { \q_no_value } }
7993 }
7994 \cs_generate_variant:Nn \prop_get:NnN { NV , No }
7995 \cs_generate_variant:Nn \prop_get:NnN { c , cV , co }

```

(End definition for `\prop_get:NnN`. This function is documented on page 129.)

`\prop_pop:NnN` Popping a value also starts by doing the split. If the key is present, save the value in the token list and update the property list as when deleting. If the key is missing, save `\q_no_value` in the token list.

```

\prop_pop:NoN
\prop_pop:cnN
\prop_pop:coN
7996 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
\prop_gpop:NnN
7997 {
\prop_gpop:NoN
7998     \__prop_split:NnTF #1 {#2}
\prop_gpop:cnN
7999     {
8000         \tl_set:Nn #3 {##2}
8001         \tl_set:Nn #1 { ##1 ##3 }
8002     }
8003     { \tl_set:Nn #3 { \q_no_value } }
8004 }
8005 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
\prop_gpop:coN
8006 {
8007     \__prop_split:NnTF #1 {#2}
8008     {
8009         \tl_set:Nn #3 {##2}
8010         \tl_gset:Nn #1 { ##1 ##3 }
8011     }
8012     { \tl_set:Nn #3 { \q_no_value } }
8013 }
8014 \cs_generate_variant:Nn \prop_pop:NnN { No }
8015 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
8016 \cs_generate_variant:Nn \prop_gpop:NnN { No }
8017 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }

```

(End definition for `\prop_pop:NnN` and `\prop_gpop:NnN`. These functions are documented on page 129.)

`\prop_item:Nn` Getting the value corresponding to a key in a property list in an expandable fashion is similar to mapping some tokens. Go through the property list one $\langle key \rangle$ – $\langle value \rangle$ pair at a time: the arguments of `__prop_item_Nn:nwn` are the $\langle key \rangle$ we are looking for, a $\langle key \rangle$ of the property list, and its associated value. The $\langle keys \rangle$ are compared (as strings). If they match, the $\langle value \rangle$ is returned, within `\exp_not:n`. The loop terminates even if the $\langle key \rangle$ is missing, and yields an empty value, because we have appended the appropriate $\langle key \rangle$ – $\langle empty value \rangle$ pair to the property list.

```

8018 \cs_new:Npn \prop_item:Nn #1#2
8019 {
8020   \exp_last_unbraced:Noo \__prop_item_Nn:nwn { \tl_to_str:n {#2} } #1
8021   \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
8022   \prg_break_point:
8023 }
8024 \cs_new:Npn \__prop_item_Nn:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
8025 {
8026   \str_if_eq_x:nnTF {#1} {#3}
8027   { \prg_break:n { \exp_not:n {#4} } }
8028   { \__prop_item_Nn:nwn {#1} }
8029 }
8030 \cs_generate_variant:Nn \prop_item:Nn { c }

```

(End definition for `\prop_item:Nn` and `__prop_item_Nn:nwn`. This function is documented on page 130.)

`\prop_pop:NnNTF` Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, `\prg_return_true:` is used after the assignments.

```

\prop_gpop:NnNTF
\prop_gpop:cnNTF
8031 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
8032 {
8033   \__prop_split:NnTF #1 {#2}
8034   {
8035     \tl_set:Nn #3 {##2}
8036     \tl_set:Nn #1 { ##1 ##3 }
8037     \prg_return_true:
8038   }
8039   { \prg_return_false: }
8040 }
8041 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
8042 {
8043   \__prop_split:NnTF #1 {#2}
8044   {
8045     \tl_set:Nn #3 {##2}
8046     \tl_gset:Nn #1 { ##1 ##3 }
8047     \prg_return_true:
8048   }
8049   { \prg_return_false: }
8050 }
8051 \prg_generate_conditional_variant:Nnn \prop_pop:NnN { c } { T , F , TF }
8052 \prg_generate_conditional_variant:Nnn \prop_gpop:NnN { c } { T , F , TF }

```

(End definition for `\prop_pop:NnNTF` and `\prop_gpop:NnNTF`. These functions are documented on page 131.)

\prop_put:Nnn

\prop_put:NnV

\prop_put:Nno

\prop_put:Nnx

\prop_put:Nvn

\prop_put:Nvv

\prop_put:Non

\prop_put:Noo

\prop_put:cnn

\prop_put:cnV

\prop_put:cno

\prop_put:cnx

\prop_put:cVn

\prop_put:cVV

\prop_put:con

\prop_put:coo

\prop_gput:Nnn

\prop_gput:NnV

\prop_gput:Nno

\prop_gput:Nnx

\prop_gput:Nvn

\prop_gput:Nvv

\prop_gput:Non

\prop_gput:Noo

\prop_gput:cnn

\prop_gput:cnV

\prop_gput:cno

\prop_gput:cnx

\prop_gput:cVn

\prop_gput:cVV

\prop_gput:con

\prop_put_if_new:Nnn

\prop_gput_if_new:Nnn

\prop_put_if_new:cnn

\prop_gput_if_new:cnn

__prop_put_if_new:NNnn

Since the branches of `__prop_split:NnTF` are used as the replacement text of an internal macro, and since the $\langle key \rangle$ and new $\langle value \rangle$ may contain arbitrary tokens, it is not safe to include them in the argument of `__prop_split:NnTF`. We thus start by storing in `\l__prop_internal_tl` tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in `__prop_split:NnTF`. If the $\langle key \rangle$ was absent, append the new key–value to the list. Otherwise concatenate the extracts `##1` and `##3` with the new key–value pair `\l__prop_internal_tl`. The updated entry is placed at the same spot as the original $\langle key \rangle$ in the property list, preserving the order of entries.

```
8053 \cs_new_protected:Npn \prop_put:Nnn { \__prop_put:NNnn \tl_set:Nx }
8054 \cs_new_protected:Npn \prop_gput:Nnn { \__prop_put:NNnn \tl_gset:Nx }
8055 \cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4
8056 {
8057   \tl_set:Nn \l__prop_internal_tl
8058   {
8059     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
8060     \s__prop { \exp_not:n {#4} }
8061   }
8062   \__prop_split:NnTF #2 {#3}
8063   { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
8064   { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
8065 }
8066 \cs_generate_variant:Nn \prop_put:Nnn
8067 { NnV , Nno , Nnx , NV , NVV , No , Noo }
8068 \cs_generate_variant:Nn \prop_gput:Nnn
8069 { c , cnV , cno , cnx , cV , cVV , co , coo }
8070 \cs_generate_variant:Nn \prop_gput:Nnn
8071 { NnV , Nno , Nnx , NV , NVV , No , Noo }
8072 \cs_generate_variant:Nn \prop_gput:Nnn
8073 { c , cnV , cno , cnx , cV , cVV , co , coo }
```

(End definition for `\prop_put:Nnn`, `\prop_gput:Nnn`, and `__prop_put:NNnn`. These functions are documented on page 129.)

Adding conditionally also splits. If the key is already present, the three brace groups given by `__prop_split:NnTF` are removed. If the key is new, then the value is added, being careful to convert the key to a string using `\tl_to_str:n`.

```
8074 \cs_new_protected:Npn \prop_put_if_new:Nnn
8075 { \__prop_put_if_new:NNnn \tl_set:Nx }
8076 \cs_new_protected:Npn \prop_gput_if_new:Nnn
8077 { \__prop_put_if_new:NNnn \tl_gset:Nx }
8078 \cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
8079 {
8080   \tl_set:Nn \l__prop_internal_tl
8081   {
8082     \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
8083     \s__prop \exp_not:n {#4} }
8084   }
8085   \__prop_split:NnTF #2 {#3}
8086   { }
8087   { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
8088 }
8089 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
8090 \cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }
```

(End definition for `\prop_put_if_new:Nnn`, `\prop_gput_if_new:Nnn`, and `__prop_put_if_new:NNnn`. These functions are documented on page 129.)

16.3 Property list conditionals

`\prop_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.
`\prop_if_exist_p:c` 8091 `\prg_new_eq_conditional:Nnn \prop_if_exist:N \cs_if_exist:N`
`\prop_if_exist:NTF` 8092 `{ TF , T , F , p }`
`\prop_if_exist:cTF` 8093 `\prg_new_eq_conditional:Nnn \prop_if_exist:c \cs_if_exist:c`
8094 `{ TF , T , F , p }`

(End definition for `\prop_if_exist:NTF`. This function is documented on page 130.)

`\prop_if_empty_p:N` Same test as for token lists.
`\prop_if_empty_p:c` 8095 `\prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }`
`\prop_if_empty:NTF` 8096 `{`
8097 `\tl_if_eq:NNTF #1 \c_empty_prop`
8098 `\prg_return_true: \prg_return_false:`
8099 `}`
`\prop_if_empty:cTF` 8100 `\prg_generate_conditional_variant:Nnn \prop_if_empty:N`
8101 `{ c } { p , T , F , TF }`

(End definition for `\prop_if_empty:NTF`. This function is documented on page 130.)

`\prop_if_in_p:Nn` Testing expandably if a key is in a property list requires to go through the key–value
`\prop_if_in_p:Nv` pairs one by one. This is rather slow, and a faster test would be
`\prop_if_in_p:No` `\prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2`
`\prop_if_in_p:cn` `{`
`\prop_if_in_p:cV` `\@@_split:NnTF #1 {#2}`
`\prop_if_in_p:co` `{ \prg_return_true: }`
`\prop_if_in:NnTF` `{ \prg_return_false: }`
`\prop_if_in:NvTF` `}`
`\prop_if_in:NoTF` but `__prop_split:NnTF` is non-expandable.
`\prop_if_in:cnTF` Instead, the key is compared to each key in turn using `\str_if_eq_x:nn`, which is
`\prop_if_in:cVTF` expandable. To terminate the mapping, we append to the property list the key that is
`\prop_if_in:coTF` searched for. This second `\tl_to_str:n` is not expanded at the start, but only when in-
`__prop_if_in:nwnn` cluded in the `\str_if_eq_x:nn`. It cannot make the breaking mechanism choke, because
`__prop_if_in:N` the arbitrary token list material is enclosed in braces. The second argument of `__prop_if_in:nwnn` is most often empty. When the *key* is found in the list, `__prop_if_in:N` receives `__prop_pair:wn`, and if it is found as the extra item, the function receives `\q_recursion_tail`, easily recognizable.

Here, `\prop_map_function:NN` is not sufficient for the mapping, since it can only map a single token, and cannot carry the key that is searched for.

8102 `\prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }`
8103 `{`
8104 `\exp_last_unbraced:Noo __prop_if_in:nwnn { \tl_to_str:n {#2} } #1`
8105 `__prop_pair:wn \tl_to_str:n {#2} \s_prop { }`
8106 `\q_recursion_tail`
8107 `\prg_break_point:`
8108 `}`
8109 `\cs_new:Npn __prop_if_in:nwnn #1#2 __prop_pair:wn #3 \s_prop #4`

```

8110 {
8111   \str_if_eq_x:nnTF {#1} {#3}
8112   { \__prop_if_in:N }
8113   { \__prop_if_in:nwn {#1} }
8114 }
8115 \cs_new:Npn \__prop_if_in:N #1
8116 {
8117   \if_meaning:w \q_recursion_tail #1
8118   \prg_return_false:
8119   \else:
8120     \prg_return_true:
8121   \fi:
8122   \prg_break:
8123 }
8124 \prg_generate_conditional_variant:Nnn \prop_if_in:Nn
8125 { NV , No , c , cV , co } { p , T , F , TF }

```

(End definition for `\prop_if_in:NnTF`, `__prop_if_in:nwn`, and `__prop_if_in:N`. This function is documented on page 130.)

16.4 Recovering values from property lists with branching

`\prop_get:NnTF` Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

```

\prop_get:NnTF
\prop_get:NVNTF
\prop_get:NoNTF
\prop_get:cnNTF
\prop_get:cVNTF
\prop_get:coNTF
8126 \prg_new_protected_conditional:Npnn \prop_get:Nn #1#2#3 { T , F , TF }
8127 {
8128   \__prop_split:NnTF #1 {#2}
8129   {
8130     \tl_set:Nn #3 {##2}
8131     \prg_return_true:
8132   }
8133   { \prg_return_false: }
8134 }
8135 \prg_generate_conditional_variant:Nnn \prop_get:NnN
8136 { NV , No , c , cV , co } { T , F , TF }

```

(End definition for `\prop_get:NnTF`. This function is documented on page 131.)

16.5 Mapping to property lists

The argument delimited by `__prop_pair:wn` is empty except at the end of the loop where it is `\prg_break:`. No need for any quark test.

```

\prop_map_function:NN
\prop_map_function:Nc
\prop_map_function:cN
\prop_map_function:cc
\__prop_map_function:Nwn
8137 \cs_new:Npn \prop_map_function:NN #1#2
8138 {
8139   \exp_after:wN \use_i_ii:nnn
8140   \exp_after:wN \__prop_map_function:Nwn
8141   \exp_after:wN #2
8142   #1
8143   \prg_break: \__prop_pair:wn \s__prop { } \prg_break_point:
8144   \prg_break_point:Nn \prop_map_break: { }
8145 }
8146 \cs_new:Npn \__prop_map_function:Nwn #1#2 \__prop_pair:wn #3 \s__prop #4
8147 {

```

```

8148     #2
8149     #1 {#3} {#4}
8150     \__prop_map_function:Nwwn #1
8151   }
8152 \cs_generate_variant:Nn \prop_map_function:NN { Nc , c , cc }

```

(End definition for `\prop_map_function:NN` and `__prop_map_function:Nwwn`. This function is documented on page 131.)

`\prop_map_inline:Nn` Mapping in line requires a nesting level counter. Store the current definition of `__prop_pair:wn`, and define it anew. At the end of the loop, revert to the earlier definition. Note that besides pairs of the form `__prop_pair:wn <key> \s__prop {<value>}`, there are a leading and a trailing tokens, but both are equal to `\scan_stop:`, hence have no effect in such inline mapping. Such `\scan_stop:` could have affected ligatures if they appeared during the mapping.

`\prop_map_inline:cn`

```

8153 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
8154 {
8155   \cs_gset_eq:cn
8156   { \__prop_map_ \int_use:N \g__kernel_prg_map_int :wn } \__prop_pair:wn
8157   \int_gincr:N \g__kernel_prg_map_int
8158   \cs_gset_protected:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
8159   #1
8160   \prg_break_point:Nn \prop_map_break:
8161   {
8162     \int_gdecr:N \g__kernel_prg_map_int
8163     \cs_gset_eq:Nc \__prop_pair:wn
8164     { \__prop_map_ \int_use:N \g__kernel_prg_map_int :wn }
8165   }
8166 }
8167 \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End definition for `\prop_map_inline:Nn`. This function is documented on page 131.)

`\prop_map_break:` The break statements are based on the general `\prg_map_break:Nn`.

`\prop_map_break:n`

```

8168 \cs_new:Npn \prop_map_break:
8169 { \prg_map_break:Nn \prop_map_break: { } }
8170 \cs_new:Npn \prop_map_break:n
8171 { \prg_map_break:Nn \prop_map_break: }

```

(End definition for `\prop_map_break:` and `\prop_map_break:n`. These functions are documented on page 132.)

16.6 Viewing property lists

`\prop_show:N` Apply the general `__kernel_chk_defined:NT` and `\msg_show:nnnnnn`. Contrarily to sequences and comma lists, we use `\msg_show_item:nn` to format both the key and the value for each pair.

`\prop_show:c`

`\prop_log:N`

`\prop_log:c`

```

8172 \cs_new_protected:Npn \prop_show:N { \__prop_show:NN \msg_show:nnxxxx }
8173 \cs_generate_variant:Nn \prop_show:N { c }
8174 \cs_new_protected:Npn \prop_log:N { \__prop_show:NN \msg_log:nnxxxx }
8175 \cs_generate_variant:Nn \prop_log:N { c }
8176 \cs_new_protected:Npn \__prop_show:NN #1#2
8177 {
8178   \__kernel_chk_defined:NT #2

```



```

8179     {
8180       #1 { LaTeX/kernel } { show-prop }
8181       { \token_to_str:N #2 }
8182       { \prop_map_function:NN #2 \msg_show_item:nn }
8183       { } { }
8184     }
8185   }

```

(End definition for `\prop_show:N` and `\prop_log:N`. These functions are documented on page 132.)

```
8186 \</initex | package>
```

17 l3msg implementation

```
8187 \*initex | package>
```

```
8188 \@@=msg
```

`\l__msg_internal_tl` A general scratch for the module.

```
8189 \tl_new:N \l__msg_internal_tl
```

(End definition for `\l__msg_internal_tl`.)

`\l__msg_line_context_bool` Controls whether the line context is shown as part of the decoration of all (non-expandable) messages.

```
8190 \bool_new:N \l__msg_line_context_bool
```

(End definition for `\l__msg_line_context_bool`.)

17.1 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

`\c__msg_text_prefix_tl` Locations for the text of messages.

`\c__msg_more_text_prefix_tl`

```
8191 \tl_const:Nn \c__msg_text_prefix_tl { msg-text~>~ }
```

```
8192 \tl_const:Nn \c__msg_more_text_prefix_tl { msg-extra~text~>~ }
```

(End definition for `\c__msg_text_prefix_tl` and `\c__msg_more_text_prefix_tl`.)

`\msg_if_exist_p:nn` Test whether the control sequence containing the message text exists or not.

`\msg_if_exist:nnTF`

```
8193 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
```

```
8194 {
```

```
8195   \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
```

```
8196   { \prg_return_true: } { \prg_return_false: }
```

```
8197 }
```

(End definition for `\msg_if_exist:nnTF`. This function is documented on page 135.)

`__msg_chk_if_free:nn` This auxiliary is similar to `__kernel_chk_if_free_cs:N`, and is used when defining messages with `\msg_new:nnnn`.

```

8198 \__kernel_patch:nnNpn { }
8199 {
8200   \__kernel_debug_log:x
8201   { Defining~message~ #1 / #2 ~\msg_line_context: }
8202 }
8203 \cs_new_protected:Npn \__msg_chk_free:nn #1#2
8204 {
8205   \msg_if_exist:nnT {#1} {#2}
8206   {
8207     \__kernel_msg_error:nxxx { kernel } { message-already-defined }
8208     {#1} {#2}
8209   }
8210 }

```

(End definition for `__msg_chk_if_free:nn`.)

`\msg_new:nnnn` Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```

\msg_new:nnnn
\msg_new:nnn
\msg_gset:nnnn
\msg_gset:nnn
\msg_set:nnnn
\msg_set:nnn
8211 \cs_new_protected:Npn \msg_new:nnnn #1#2
8212 {
8213   \__msg_chk_free:nn {#1} {#2}
8214   \msg_gset:nnnn {#1} {#2}
8215 }
8216 \cs_new_protected:Npn \msg_new:nnn #1#2#3
8217 { \msg_new:nnnn {#1} {#2} {#3} { } }
8218 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
8219 {
8220   \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
8221   ##1##2##3##4 {#3}
8222   \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
8223   ##1##2##3##4 {#4}
8224 }
8225 \cs_new_protected:Npn \msg_set:nnn #1#2#3
8226 { \msg_set:nnnn {#1} {#2} {#3} { } }
8227 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
8228 {
8229   \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
8230   ##1##2##3##4 {#3}
8231   \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
8232   ##1##2##3##4 {#4}
8233 }
8234 \cs_new_protected:Npn \msg_gset:nnn #1#2#3
8235 { \msg_gset:nnnn {#1} {#2} {#3} { } }

```

(End definition for `\msg_new:nnnn` and others. These functions are documented on page 134.)

17.2 Messages: support functions and text

`\c__msg_coding_error_text_tl` Simple pieces of text for messages.

```

\c__msg_continue_text_tl
\c__msg_critical_text_tl
\c__msg_fatal_text_tl
\c__msg_help_text_tl
\c__msg_no_info_text_tl
\c__msg_on_line_text_tl
\c__msg_return_text_tl
\c__msg_trouble_text_tl
8236 \tl_const:Nn \c__msg_coding_error_text_tl
8237 {
8238   This~is~a~coding~error.

```

```

8239     \\\ \\\
8240   }
8241   \tl_const:Nn \c__msg_continue_text_tl
8242     { Type~<return>~to~continue }
8243   \tl_const:Nn \c__msg_critical_text_tl
8244     { Reading~the~current~file~'\g_file_curr_name_str'~will~stop. }
8245   \tl_const:Nn \c__msg_fatal_text_tl
8246     { This~is~a~fatal~error:~LaTeX~will~abort. }
8247   \tl_const:Nn \c__msg_help_text_tl
8248     { For~immediate~help~type~H~<return> }
8249   \tl_const:Nn \c__msg_no_info_text_tl
8250     {
8251       LaTeX~does~not~know~anything~more~about~this~error,~sorry.
8252       \c__msg_return_text_tl
8253     }
8254   \tl_const:Nn \c__msg_on_line_text_tl { on-line }
8255   \tl_const:Nn \c__msg_return_text_tl
8256     {
8257       \\\ \\\
8258       Try~typing~<return>~to~proceed.
8259       \\\
8260       If~that~doesn't~work,~type~X~<return>~to~quit.
8261     }
8262   \tl_const:Nn \c__msg_trouble_text_tl
8263     {
8264       \\\ \\\
8265       More~errors~will~almost~certainly~follow: \\\
8266       the~LaTeX~run~should~be~aborted.
8267     }

```

(End definition for `\c__msg_coding_error_text_tl` and others.)

`\msg_line_number:` For writing the line number nicely. `\msg_line_context:` was set up earlier, so this is not new.

```

\msg_line_context:
8268 \cs_new:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
8269 \cs_gset:Npn \msg_line_context:
8270   {
8271     \c__msg_on_line_text_tl
8272     \c_space_tl
8273     \msg_line_number:
8274   }

```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 135.)

17.3 Showing messages: low level mechanism

`\msg_interrupt:nnn` The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup T_EX's `\errhelp` register before issuing an `\errmessage`.

```

8275 \cs_new_protected:Npn \msg_interrupt:nnn #1#2#3
8276   {
8277     \tl_if_empty:nTF {#3}

```

```

8278     {
8279       \_msg_interrupt_wrap:nn { \ \ \c__msg_no_info_text_tl }
8280       {#1 \ \ \ \ #2 \ \ \ \ \c__msg_continue_text_tl }
8281     }
8282     {
8283       \_msg_interrupt_wrap:nn { \ \ \ #3 }
8284       {#1 \ \ \ \ #2 \ \ \ \ \c__msg_help_text_tl }
8285     }
8286   }

```

(End definition for `\msg_interrupt:nnn`. This function is documented on page 139.)

```

\_msg_interrupt_wrap:nn
\_msg_interrupt_more_text:n

```

First setup TeX's `\errhelp` register with the extra help #1, then build a nice-looking error message with #2. Everything is done using x-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary `_msg_interrupt_more_text:n` receives its argument as a line-wrapped string, which is thus unaffected by expansion.

```

8287 \cs_new_protected:Npn \_msg_interrupt_wrap:nn #1#2
8288   {
8289     \iow_wrap:nnnN {#1} { | ~ } { } \_msg_interrupt_more_text:n
8290     \iow_wrap:nnnN {#2} { ! ~ } { } \_msg_interrupt_text:n
8291   }
8292 \cs_new_protected:Npn \_msg_interrupt_more_text:n #1
8293   {
8294     \exp_args:Nx \tex_errhelp:D
8295     {
8296       |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
8297       #1 \iow_newline:
8298       |.....
8299     }
8300   }

```

(End definition for `_msg_interrupt_wrap:nn` and `_msg_interrupt_more_text:n`.)

```

\_msg_interrupt_text:n

```

The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made “invisible”: TeX's own information involves the macro in which `\errmessage` is called, and the end of the argument of the `\errmessage`, including the closing brace. We use an active `!` to call the `\errmessage` primitive, and end its argument with `\use_none:n {<dots>}` which fills the output with dots. Two trailing closing braces are turned into spaces to hide them as well. The group in which we alter the definition of the active `!` is closed before producing the message: this ensures that tokens inserted by typing `I` in the command-line are inserted after the message is entirely cleaned up.

The `_kernel_iow_with:Nnn` auxiliary, defined in `l3file`, expects an *<integer variable>*, an integer *<value>*, and some *<code>*. It runs the *<code>* after ensuring that the *<integer variable>* takes the given *<value>*, then restores the former value of the *<integer variable>* if needed. We use it to ensure that the `\newlinechar` is 10, as needed for `\iow_newline:` to work, and that `\errorcontextlines` is `-1`, to avoid showing irrelevant context. Note that restoring the former value of these integers requires inserting tokens after the `\errmessage`, which go in the way of tokens which could be inserted by the user. This is unavoidable.

```

8301 \group_begin:
8302   \char_set_lccode:nn {\ } {\ }

```

```

8303 \char_set_lccode:nn {'\}} {'\ }
8304 \char_set_lccode:nn {'&} {'\!}
8305 \char_set_catcode_active:N \&
8306 \tex_lowercase:D
8307 {
8308   \group_end:
8309   \cs_new_protected:Npn \_msg_interrupt_text:n #1
8310   {
8311     \iow_term:x
8312     {
8313       \iow_newline:
8314       !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
8315       \iow_newline:
8316       !
8317     }
8318     \_kernel_iow_with:Nnn \tex_newlinechar:D { '\^^J }
8319     {
8320       \_kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
8321       {
8322         \group_begin:
8323         \cs_set_protected:Npn &
8324         {
8325           \tex_errmessage:D
8326           {
8327             #1
8328             \use_none:n
8329             { ..... }
8330           }
8331         }
8332         \exp_after:wN
8333         \group_end:
8334         &
8335       }
8336     }
8337   }
8338 }

```

(End definition for _msg_interrupt_text:n.)

\msg_log:n Printing to the log or terminal without a stop is rather easier. A bit of simple visual
\msg_term:n work sets things off nicely.

```

8339 \cs_new_protected:Npn \msg_log:n #1
8340 {
8341   \iow_log:n { ..... }
8342   \iow_wrap:nnnN { . ~ #1 } { . ~ } { } \iow_log:n
8343   \iow_log:n { ..... }
8344 }
8345 \cs_new_protected:Npn \msg_term:n #1
8346 {
8347   \iow_term:n { ***** }
8348   \iow_wrap:nnnN { * ~ #1 } { * ~ } { } \iow_term:n
8349   \iow_term:n { ***** }
8350 }

```

(End definition for \msg_log:n and \msg_term:n. These functions are documented on page 140.)

17.4 Displaying messages

L^AT_EX is handling error messages and so the T_EX ones are disabled. This is already done by the L^AT_EX 2_ε kernel, so to avoid messing up any deliberate change by a user this is only set in format mode.

```
8351 (*initex)
8352 \int_gset:Nn \tex_errorcontextlines:D { -1 }
8353 \end{initex}
```

\msg_fatal_text:n A function for issuing messages: both the text and order could in principle vary.

```
\msg_critical_text:n
\msg_error_text:n
\msg_warning_text:n
\msg_info_text:n
8354 \cs_new:Npn \msg_fatal_text:n #1
8355 {
8356   Fatal~#1~error
8357   \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
8358 }
8359 \cs_new:Npn \msg_critical_text:n #1
8360 {
8361   Critical~#1~error
8362   \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
8363 }
8364 \cs_new:Npn \msg_error_text:n #1
8365 {
8366   #1~error
8367   \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
8368 }
8369 \cs_new:Npn \msg_warning_text:n #1
8370 {
8371   #1~warning
8372   \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
8373 }
8374 \cs_new:Npn \msg_info_text:n #1
8375 {
8376   #1~info
8377   \bool_if:NT \l__msg_line_context_bool { ~ \msg_line_context: }
8378 }
```

(End definition for `\msg_fatal_text:n` and others. These functions are documented on page 135.)

\msg_see_documentation_text:n Contextual footer information. The L^AT_EX module only comprises L^AT_EX3 code, so we refer to the L^AT_EX3 documentation rather than simply “L^AT_EX”.

```
8379 \cs_new:Npn \msg_see_documentation_text:n #1
8380 {
8381   \ \ \ See~the~
8382   \str_if_eq:nnTF {#1} { LaTeX } { LaTeX3 } {#1} ~
8383   documentation~for~further~information.
8384 }
```

(End definition for `\msg_see_documentation_text:n`. This function is documented on page 136.)

__msg_class_new:nn

```
8385 \group_begin:
8386 \cs_set_protected:Npn \__msg_class_new:nn #1#2
8387 {
8388   \prop_new:c { l__msg_redirect_ #1 _prop }
```

```

8389 \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn }
8390   ##1##2##3##4##5##6 {#2}
8391 \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
8392 {
8393   \use:x
8394   {
8395     \exp_not:n { \_msg_use:nnnnnn {#1} {##1} {##2} }
8396     { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
8397     { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
8398   }
8399 }
8400 \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
8401 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
8402 \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
8403 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
8404 \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
8405 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
8406 \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
8407 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
8408 \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5##6
8409 {
8410   \use:x
8411   {
8412     \exp_not:N \exp_not:n
8413     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
8414     {##3} {##4} {##5} {##6}
8415   }
8416 }
8417 \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
8418 { \exp_not:c { msg_ #1 :nnxxx } {##1} {##2} {##3} {##4} {##5} { } }
8419 \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
8420 { \exp_not:c { msg_ #1 :nnxxx } {##1} {##2} {##3} {##4} { } { } }
8421 \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
8422 { \exp_not:c { msg_ #1 :nnxxx } {##1} {##2} {##3} { } { } { } }
8423 }

```

(End definition for `_msg_class_new:nn`.)

`\msg_fatal:nnnnnn` For fatal errors, after the error message `TeX` bails out.

```

8424 \msg_fatal:nnxxx \_msg_class_new:nn { fatal }
8425 {
8426   \msg_interrupt:nnn
8427   { \msg_fatal_text:n {#1} : ~ "#2" }
8428   {
8429     \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
8430     \msg_see_documentation_text:n {#1}
8431   }
8432   { \c__msg_fatal_text_tl }
8433   \tex_end:D
8434 }

```

(End definition for `\msg_fatal:nnnnnn` and others. These functions are documented on page 136.)

`\msg_critical:nnnnnn` Not quite so bad: just end the current file.

```

\msg_critical:nnxxx
\msg_critical:nnnnn
\msg_critical:nnxxx
\msg_critical:nnnn
\msg_critical:nnxx
\msg_critical:nnn
\msg_critical:nnx
\msg_critical:nn

```

```

8435 \__msg_class_new:nn { critical }
8436 {
8437   \msg_interrupt:nnn
8438   { \msg_critical_text:n {#1} : ~ "#2" }
8439   {
8440     \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
8441     \msg_see_documentation_text:n {#1}
8442   }
8443   { \c__msg_critical_text_tl }
8444   \tex_endinput:D
8445 }

```

(End definition for `\msg_critical:nnnnnn` and others. These functions are documented on page 136.)

`\msg_error:nnnnnn` For an error, the interrupt routine is called. We check if there is a “more text” by
`\msg_error:nnxxxx` comparing that control sequence with a permanently empty text.
`\msg_error:nnnnnn` 8446 __msg_class_new:nn { error }
`\msg_error:nnxxx` 8447 {
`\msg_error:nnnn` 8448 __msg_error:cnnnnn
`\msg_error:nnxx` 8449 { \c__msg_more_text_prefix_tl #1 / #2 }
`\msg_error:nnn` 8450 {#3} {#4} {#5} {#6}
`\msg_error:nnx` 8451 {
`\msg_error:nn` 8452 \msg_interrupt:nnn
8453 { \msg_error_text:n {#1} : ~ "#2" }
8454 {
8455 \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
8456 \msg_see_documentation_text:n {#1}
8457 }
8458 }
8459 }
8460 \cs_new_protected:Npn __msg_error:cnnnnn #1#2#3#4#5#6
8461 {
8462 \cs_if_eq:cNTF {#1} __msg_no_more_text:nnnn
8463 { #6 { } }
8464 { #6 { \use:c {#1} {#2} {#3} {#4} {#5} } }
8465 }
8466 \cs_new:Npn __msg_no_more_text:nnnn #1#2#3#4 { }

(End definition for `\msg_error:nnnnnn` and others. These functions are documented on page 137.)

`\msg_warning:nnnnnn` Warnings are printed to the terminal.
`\msg_warning:nnxxxx` 8467 __msg_class_new:nn { warning }
`\msg_warning:nnnnn` 8468 {
`\msg_warning:nnxxx` 8469 \msg_term:n
`\msg_warning:nnnn` 8470 {
8471 \msg_warning_text:n {#1} : ~ "#2" \\ \\
8472 \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
8473 }
8474 }
`\msg_warning:nnx`
`\msg_warning:nn`

(End definition for `\msg_warning:nnnnnn` and others. These functions are documented on page 137.)

`\msg_info:nnnnnn` Information only goes into the log.
`\msg_info:nnxxxx` 8475 __msg_class_new:nn { info }
`\msg_info:nnnnn` 8476 {
`\msg_info:nnxxx`
`\msg_info:nnnn`
`\msg_info:nnxx`
`\msg_info:nnn`
`\msg_info:nnx`
`\msg_info:nn`


```

8477     \msg_log:n
8478     {
8479         \msg_info_text:n {#1} : ~ "#2" \\ \\
8480         \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
8481     }
8482 }

```

(End definition for \msg_info:nnnnnn and others. These functions are documented on page 137.)

\msg_log:nnnnnn “Log” data is very similar to information, but with no extras added.

```

\msg_log:nnxxxx 8483 \__msg_class_new:nn { log }
\msg_log:nnnnnn 8484 {
\msg_log:nnxxx 8485     \iow_wrap:nnnN
\msg_log:nnnn 8486     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\msg_log:nnxx 8487     { } { } \iow_log:n
\msg_log:nnn 8488 }

```

\msg_log:nnx (End definition for \msg_log:nnnnnn and others. These functions are documented on page 137.)

\msg_log:n

\msg_none:nnnnnn The none message type is needed so that input can be gobbled.

```

\msg_none:nnxxxx 8489 \__msg_class_new:nn { none } { }

```

\msg_none:nnnnnn (End definition for \msg_none:nnnnnn and others. These functions are documented on page 138.)

\msg_none:nnxxx

\msg_show:nnnnnn

The show message type is used for \seq_show:N and similar complicated data structures.

\msg_show:nnxxx Wrap the given text with a trailing dot (important later) then pass it to __msg_show:n.
\msg_show:nnnn If there is \\>~ (or if the whole thing starts with >~) we split there, print the first part
\msg_show:nnnnn and show the second part using \showtokens (the \exp_after:wN ensure a nice display).
\msg_show:nnxxx Note that that primitive adds a leading >~ and trailing dot. That is why we included a
\msg_show:nn trailing dot before wrapping and removed it afterwards. If there is no \\>~ do the same
\msg_show:nnn but with an empty second part which adds a spurious but inevitable >~.

```

\msg_show:nnx 8490 \__msg_class_new:nn { show }
\msg_show:nn 8491 {
\__msg_show:n 8492     \iow_wrap:nnnN
\__msg_show:w 8493     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
\__msg_show_dot:w 8494     { } { } \__msg_show:n
\__msg_show:nn 8495 }
8496 \cs_new_protected:Npn \__msg_show:n #1
8497 {
8498     \tl_if_in:nnTF { ^^J #1 } { ^^J > ~ }
8499     {
8500         \tl_if_in:nnTF { #1 \q_mark } { . \q_mark }
8501         { \__msg_show_dot:w } { \__msg_show:w }
8502         ^^J #1 \q_stop
8503     }
8504     { \__msg_show:nn { ? #1 } { } }
8505 }
8506 \cs_new:Npn \__msg_show_dot:w #1 ^^J > ~ #2 . \q_stop
8507 { \__msg_show:nn {#1} {#2} }
8508 \cs_new:Npn \__msg_show:w #1 ^^J > ~ #2 \q_stop
8509 { \__msg_show:nn {#1} {#2} }
8510 \cs_new_protected:Npn \__msg_show:nn #1#2
8511 {
8512     \tl_if_empty:nF {#1}

```

```

8513     { \exp_args:No \iow_term:n { \use_none:n #1 } }
8514 \tl_set:Nn \l__msg_internal_tl {#2}
8515 \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
8516 {
8517     \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
8518     {
8519         \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
8520         { \exp_after:wN \l__msg_internal_tl }
8521     }
8522 }
8523 }

```

(End definition for `\msg_show:nnnnnn` and others. These functions are documented on page 239.)

End the group to eliminate `__msg_class_new:nn`.

```
8524 \group_end:
```

`__msg_class_chk_exist:nT` Checking that a message class exists. We build this from `\cs_if_free:cTF` rather than `\cs_if_exist:cTF` because that avoids reading the second argument earlier than necessary.

```

8525 \cs_new:Npn \__msg_class_chk_exist:nT #1
8526 {
8527     \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
8528     { \__kernel_msg_error:nx { kernel } { message-class-unknown } {#1} }
8529 }

```

(End definition for `__msg_class_chk_exist:nT`.)

`\l__msg_class_tl` Support variables needed for the redirection system.

```

\l__msg_current_class_tl 8530 \tl_new:N \l__msg_class_tl
8531 \tl_new:N \l__msg_current_class_tl

```

(End definition for `\l__msg_class_tl` and `\l__msg_current_class_tl`.)

`\l__msg_redirect_prop` For redirection of individually-named messages

```
8532 \prop_new:N \l__msg_redirect_prop
```

(End definition for `\l__msg_redirect_prop`.)

`\l__msg_hierarchy_seq` During redirection, split the message name into a sequence: `{/module/submodule}`, `{/module}`, and `{}`.

```
8533 \seq_new:N \l__msg_hierarchy_seq
```

(End definition for `\l__msg_hierarchy_seq`.)

`\l__msg_class_loop_seq` Classes encountered when following redirections to check for loops.

```
8534 \seq_new:N \l__msg_class_loop_seq
```

(End definition for `\l__msg_class_loop_seq`.)

```

    \_msg\_use:nnnnnnn
    \_msg\_use\_redirect\_name:n
    \_msg\_use\_hierarchy:nwN
    \_msg\_use\_redirect\_module:n
    \_msg\_use\_code:

```

Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around. The assignment to `_msg_use_code:` is similar to `\tl_set:Nn`. The message is eventually produced with whatever `\l_msg_class_tl` is when `_msg_use_code:` is called. Here is also a good place to suppress tracing output if the trace package is loaded since all (non-expandable) messages go through this auxiliary.

```

8535 \cs_new_protected:Npn \_msg\_use:nnnnnnn #1#2#3#4#5#6#7
8536 {
8537   <package> \use:c { conditionally@traceoff }
8538   \msg_if_exist:nnTF {#2} {#3}
8539   {
8540     \_msg\_class\_chk\_exist:nT {#1}
8541     {
8542       \tl\_set:Nn \l\_msg\_current\_class\_tl {#1}
8543       \cs\_set\_protected:Npx \_msg\_use\_code:
8544       {
8545         \exp\_not:n
8546         {
8547           \use:c { \_msg\_ \l\_msg\_class\_tl \_code:nnnnnnn }
8548             {#2} {#3} {#4} {#5} {#6} {#7}
8549         }
8550       }
8551       \_msg\_use\_redirect\_name:n { #2 / #3 }
8552     }
8553   }
8554   { \_kernel\_msg\_error:nnxx { kernel } { message-unknown } {#2} {#3} }
8555   <package> \use:c { conditionally@traceon }
8556 }
8557 \cs\_new\_protected:Npn \_msg\_use\_code: { }

```

The first check is for a individual message redirection. If this applies then no further redirection is attempted. Otherwise, split the message name into `<module>`, `<submodule>` and `<message>` (with an arbitrary number of slashes), and store `{/module/submodule}`, `{/module}` and `{}` into `\l_msg_hierarchy_seq`. We then map through this sequence, applying the most specific redirection.

```

8558 \cs\_new\_protected:Npn \_msg\_use\_redirect\_name:n #1
8559 {
8560   \prop\_get:NnNTF \l\_msg\_redirect\_prop { / #1 } \l\_msg\_class\_tl
8561   { \_msg\_use\_code: }
8562   {
8563     \seq\_clear:N \l\_msg\_hierarchy\_seq
8564     \_msg\_use\_hierarchy:nwN { }
8565     #1 \q\_mark \_msg\_use\_hierarchy:nwN
8566     / \q\_mark \use\_none\_delimit\_by\_q\_stop:w
8567     \q\_stop
8568     \_msg\_use\_redirect\_module:n { }
8569   }
8570 }
8571 \cs\_new\_protected:Npn \_msg\_use\_hierarchy:nwN #1#2 / #3 \q\_mark #4
8572 {
8573   \seq\_put\_left:Nn \l\_msg\_hierarchy\_seq {#1}
8574   #4 { #1 / #2 } #3 \q\_mark #4
8575 }

```

At this point, the items of `\l__msg_hierarchy_seq` are the various levels at which we should look for a redirection. Redirections which are less specific than the argument of `__msg_use_redirect_module:n` are not attempted. This argument is empty for a class redirection, `/module` for a module redirection, *etc.* Loop through the sequence to find the most specific redirection, with module `##1`. The loop is interrupted after testing for a redirection for `##1` equal to the argument `#1` (least specific redirection allowed). When a redirection is found, break the mapping, then if the redirection targets the same class, output the code with that class, and otherwise set the target as the new current class, and search for further redirections. Those redirections should be at least as specific as `##1`.

```

8576 \cs_new_protected:Npn \__msg_use_redirect_module:n #1
8577 {
8578   \seq_map_inline:Nn \l__msg_hierarchy_seq
8579   {
8580     \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
8581     {##1} \l__msg_class_tl
8582     {
8583       \seq_map_break:n
8584       {
8585         \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
8586         { \__msg_use_code: }
8587         {
8588           \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
8589           \__msg_use_redirect_module:n {##1}
8590         }
8591       }
8592     }
8593     {
8594       \str_if_eq:nnT {##1} {#1}
8595       {
8596         \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
8597         \seq_map_break:n { \__msg_use_code: }
8598       }
8599     }
8600   }
8601 }

```

(End definition for `__msg_use:nnnnnnn` and others.)

`\msg_redirect_name:nnn` Named message always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```

8602 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
8603 {
8604   \tl_if_empty:nTF {#3}
8605   { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
8606   {
8607     \__msg_class_chk_exist:nT {#3}
8608     { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
8609   }
8610 }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page [139](#).)

```

\msg_redirect_class:nn
\msg_redirect_module:nnn
  \__msg_redirect:nnn
\__msg_redirect_loop_chk:nnn
\__msg_redirect_loop_list:n

```

If the target class is empty, eliminate the corresponding redirection. Otherwise, add the redirection. We must then check for a loop: as an initialization, we start by storing the initial class in \l__msg_current_class_tl.

```

8611 \cs_new_protected:Npn \msg_redirect_class:nn
8612   { \__msg_redirect:nnn { } }
8613 \cs_new_protected:Npn \msg_redirect_module:nnn #1
8614   { \__msg_redirect:nnn { / #1 } }
8615 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
8616   {
8617     \__msg_class_chk_exist:nT {#2}
8618     {
8619       \tl_if_empty:nTF {#3}
8620       { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
8621       {
8622         \__msg_class_chk_exist:nT {#3}
8623         {
8624           \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
8625           \tl_set:Nn \l__msg_current_class_tl {#2}
8626           \seq_clear:N \l__msg_class_loop_seq
8627           \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
8628         }
8629       }
8630     }
8631   }

```

Since multiple redirections can only happen with increasing specificity, a loop requires that all steps are of the same specificity. The new redirection can thus only create a loop with other redirections for the exact same module, #1, and not submodules. After some initialization above, follow redirections with \l__msg_class_tl, and keep track in \l__msg_class_loop_seq of the various classes encountered. A redirection from a class to itself, or the absence of redirection both mean that there is no loop. A redirection to the initial class marks a loop. To break it, we must decide which redirection to cancel. The user most likely wants the newly added redirection to hold with no further redirection. We thus remove the redirection starting from #2, target of the new redirection. Note that no message is emitted by any of the underlying functions: otherwise we may get an infinite loop because of a message from the message system itself.

```

8632 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
8633   {
8634     \seq_put_right:Nn \l__msg_class_loop_seq {#1}
8635     \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
8636     {
8637       \str_if_eq_x:nnF { \l__msg_class_tl } {#1}
8638       {
8639         \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
8640         {
8641           \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
8642           \__kernel_msg_warning:nnxxxx
8643           { kernel } { message-redirect-loop }
8644           { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
8645           { \seq_item:Nn \l__msg_class_loop_seq { 2 } }
8646           {#3}
8647           {
8648             \seq_map_function:NN \l__msg_class_loop_seq

```

```

8649         \_msg_redirect_loop_list:n
8650         { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
8651     }
8652 }
8653 { \_msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
8654 }
8655 }
8656 }
8657 \cs_generate_variant:Nn \_msg_redirect_loop_chk:nnn { o }
8658 \cs_new:Npn \_msg_redirect_loop_list:n #1 { {#1} ~ => ~ }

```

(End definition for `\msg_redirect_class:nn` and others. These functions are documented on page 138.)

17.5 Kernel-specific functions

`_kernel_msg_new:nnnn` The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

```

\_kernel_msg_new:nnnn
\_kernel_msg_new:nnn
\_kernel_msg_set:nnnn
\_kernel_msg_set:nnn
8659 \cs_new_protected:Npn \_kernel_msg_new:nnnn #1#2
8660 { \msg_new:nnnn { LaTeX } { #1 / #2 } }
8661 \cs_new_protected:Npn \_kernel_msg_new:nnn #1#2
8662 { \msg_new:nnn { LaTeX } { #1 / #2 } }
8663 \cs_new_protected:Npn \_kernel_msg_set:nnnn #1#2
8664 { \msg_set:nnnn { LaTeX } { #1 / #2 } }
8665 \cs_new_protected:Npn \_kernel_msg_set:nnn #1#2
8666 { \msg_set:nnn { LaTeX } { #1 / #2 } }

```

(End definition for `_kernel_msg_new:nnnn` and others.)

`_msg_kernel_class_new:nN` All the functions for kernel messages come in variants ranging from 0 to 4 arguments. Those with less than 4 arguments are defined in terms of the 4-argument variant, in a way very similar to `_msg_class_new:nn`. This auxiliary is destroyed at the end of the group.

```

8667 \group_begin:
8668   \cs_set_protected:Npn \_msg_kernel_class_new:nN #1
8669   { \_msg_kernel_class_new_aux:nN { __kernel_msg_ #1 } }
8670   \cs_set_protected:Npn \_msg_kernel_class_new_aux:nN #1#2
8671   {
8672     \cs_new_protected:cpn { #1 :nnnnnn } ##1##2##3##4##5##6
8673     {
8674       \use:x
8675       {
8676         \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }
8677         { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
8678         { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
8679       }
8680     }
8681     \cs_new_protected:cpx { #1 :nnnnn } ##1##2##3##4##5
8682     { \exp_not:c { #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
8683     \cs_new_protected:cpx { #1 :nnnn } ##1##2##3##4
8684     { \exp_not:c { #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
8685     \cs_new_protected:cpx { #1 :nnn } ##1##2##3
8686     { \exp_not:c { #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
8687     \cs_new_protected:cpx { #1 :nn } ##1##2

```

```

8688 { \exp_not:c { #1 :nnnnnn } {##1} {##2} { } { } { } }
8689 \cs_new_protected:cpx { #1 :nnxxxx } ##1##2##3##4##5##6
8690 {
8691   \use:x
8692   {
8693     \exp_not:N \exp_not:n
8694     { \exp_not:c { #1 :nnnnnn } {##1} {##2} }
8695     {##3} {##4} {##5} {##6}
8696   }
8697 }
8698 \cs_new_protected:cpx { #1 :nnxxx } ##1##2##3##4##5
8699 { \exp_not:c { #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
8700 \cs_new_protected:cpx { #1 :nnxx } ##1##2##3##4
8701 { \exp_not:c { #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
8702 \cs_new_protected:cpx { #1 :nnx } ##1##2##3
8703 { \exp_not:c { #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
8704 }

```

Neither fatal kernel errors nor kernel errors can be redirected. We directly use the code for (non-kernel) fatal errors and errors, adding the “`LATEX`” module name. Three functions are already defined by `l3basics`; we need to undefine them to avoid errors.

(End definition for `_kernel_msg_fatal:nnnnnn` and others.)

```
8710 \_msg_kernel_class_new:nN { warning } \msg_warning:nnxxxxx
8711 \_msg_kernel_class_new:nN { info } \msg_info:nnxxxxx
```

End the group to eliminate _msg_kernel_class_new:nN.

Error messages needed to actually implement the message system itself.

```

8727 \c__msg_return_text_tl
8728 }
8729 \__kernel_msg_new:nnnn { kernel } { message-class-unknown }
8730 { Unknown~message~class~'~#1'~. }
8731 {
8732 LaTeX~has~been~asked~to~redirect~messages~to~a~class~'~#1':~\\
8733 this~was~never~defined.
8734 \c__msg_return_text_tl
8735 }
8736 \__kernel_msg_new:nnnn { kernel } { message-redirect-loop }
8737 {
8738 Message~redirection~loop~caused~by~ {#1} ~>~ {#2}
8739 \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
8740 }
8741 {
8742 Adding~the~message~redirection~ {#1} ~>~ {#2}
8743 \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
8744 created~an~infinite~loop~\\\
8745 \iow_indent:n { #4 \\\ }
8746 }

```

Messages for earlier kernel modules plus a few for l3keys which cover coding errors.

```

8747 \__kernel_msg_new:nnnn { kernel } { bad-number-of-arguments }
8748 { Function~'~#1'~cannot~be~defined~with~#2~arguments. }
8749 {
8750 \c__msg_coding_error_text_tl
8751 LaTeX~has~been~asked~to~define~a~function~'~#1'~with~
8752 #2~arguments~.~
8753 TeX~allows~between~0~and~9~arguments~for~a~single~function.
8754 }
8755 \__kernel_msg_new:nnn { kernel } { char-active }
8756 { Cannot~generate~active~chars. }
8757 \__kernel_msg_new:nnn { kernel } { char-invalid-catcode }
8758 { Invalid~catcode~for~char~generation. }
8759 \__kernel_msg_new:nnn { kernel } { char-null-space }
8760 { Cannot~generate~null~char~as~a~space. }
8761 \__kernel_msg_new:nnn { kernel } { char-out-of-range }
8762 { Charcode~requested~out~of~engine~range. }
8763 \__kernel_msg_new:nnn { kernel } { char-space }
8764 { Cannot~generate~space~chars. }
8765 \__kernel_msg_new:nnnn { kernel } { command-already-defined }
8766 { Control~sequence~#1~already~defined. }
8767 {
8768 \c__msg_coding_error_text_tl
8769 LaTeX~has~been~asked~to~create~a~new~control~sequence~'~#1'~
8770 but~this~name~has~already~been~used~elsewhere. \\ \\
8771 The~current~meaning~is:\\
8772 \\ #2
8773 }
8774 \__kernel_msg_new:nnnn { kernel } { command-not-defined }
8775 { Control~sequence~#1~undefined. }
8776 {
8777 \c__msg_coding_error_text_tl
8778 LaTeX~has~been~asked~to~use~a~control~sequence~'~#1':~\\
8779 this~has~not~been~defined~yet.

```



```

8780 }
8781 \_kernel_msg_new:nnn { kernel } { deprecated-command }
8782 {
8783   The-deprecated-command~'#2'~has-been-or~will-be-removed-on~#1.
8784   \tl_if_empty:nF {#3} { ~Use-instead~'#3'. }
8785 }
8786 \_kernel_msg_new:nnnn { kernel } { empty-search-pattern }
8787 { Empty-search-pattern. }
8788 {
8789   \c_msg_coding_error_text_tl
8790   LaTeX-has-been-asked-to-replace-an-empty-pattern-by~'#1':~that~
8791   would-lead-to-an-infinite-loop!
8792 }
8793 \_kernel_msg_new:nnnn { kernel } { out-of-registers }
8794 { No-room-for-a-new~#1. }
8795 {
8796   TeX-only-supports~\int_use:N \c_max_register_int \ %
8797   of-each-type.~All-the~#1~registers-have-been-used.~
8798   This-run-will-be-aborted-now.
8799 }
8800 \_kernel_msg_new:nnnn { kernel } { non-base-function }
8801 { Function~'#1'~is-not-a-base-function }
8802 {
8803   \c_msg_coding_error_text_tl
8804   Functions-defined-through~\iow_char:N\cs_new:Nn~must-have~
8805   a-signature-consisting-of-only-normal-arguments~'N'~and~'n'.~
8806   To-define-variants-use~\iow_char:N\cs_generate_variant:Nn~
8807   and-to-define-other-functions-use~\iow_char:N\cs_new:Npn.
8808 }
8809 \_kernel_msg_new:nnnn { kernel } { missing-colon }
8810 { Function~'#1'~contains-no~':'~. }
8811 {
8812   \c_msg_coding_error_text_tl
8813   Code-level-functions-must-contain~':'~to-separate-the~
8814   argument-specification-from-the-function-name.~This-is~
8815   needed-when-defining-conditionals-or-variants,~or-when-building-a~
8816   parameter-text-from-the-number-of-arguments-of-the-function.
8817 }
8818 \_kernel_msg_new:nnnn { kernel } { overflow }
8819 { Integers-larger-than~230-1~cannot-be-stored-in-arrays. }
8820 {
8821   An-attempt-was-made-to-store~#3-at-position~#2-in-the-array~'#1'.~
8822   The-largest-allowed-value~#4~will-be-used-instead.
8823 }
8824 \_kernel_msg_new:nnnn { kernel } { out-of-bounds }
8825 { Access-to-an-entry-beyond-an-array's-bounds. }
8826 {
8827   An-attempt-was-made-to-access-or-store-data-at-position~#2-of-the~
8828   array~'#1',~but-this-array-has-entries-at-positions-from~1~to~#3.
8829 }
8830 \_kernel_msg_new:nnnn { kernel } { protected-predicate }
8831 { Predicate~'#1'~must-be-expandable. }
8832 {
8833   \c_msg_coding_error_text_tl

```

```

8834 LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate.~
8835 Only~expandable~tests~can~have~a~predicate~version.
8836 }
8837 \__kernel_msg_new:nnn { kernel } { randint-backward-range }
8838 { Bounds~ordered~backwards~in~\int_rand:nn {#1}~{#2}. }
8839 \__kernel_msg_new:nnnn { kernel } { conditional-form-unknown }
8840 { Conditional~form~'#1'~for~function~'#2'~unknown. }
8841 {
8842 \c_msg_coding_error_text_tl
8843 LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
8844 the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
8845 }
8846 \__kernel_msg_new:nnnn { kernel } { key-no-property }
8847 { No~property~given~in~definition~of~key~'#1'. }
8848 {
8849 \c_msg_coding_error_text_tl
8850 Inside~\keys_define:nn each~key~name~
8851 needs~a~property: \ \ \
8852 \iow_indent:n { #1 .<property> } \ \ \
8853 LaTeX~did~not~find~a~'.'~to~indicate~the~start~of~a~property.
8854 }
8855 \__kernel_msg_new:nnnn { kernel } { key-property-boolean-values-only }
8856 { The~property~'#1'~accepts~boolean~values~only. }
8857 {
8858 \c_msg_coding_error_text_tl
8859 The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
8860 }
8861 \__kernel_msg_new:nnnn { kernel } { key-property-requires-value }
8862 { The~property~'#1'~requires~a~value. }
8863 {
8864 \c_msg_coding_error_text_tl
8865 LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'.\ \
8866 No~value~was~given~for~the~property,~and~one~is~required.
8867 }
8868 \__kernel_msg_new:nnnn { kernel } { key-property-unknown }
8869 { The~key~property~'#1'~is~unknown. }
8870 {
8871 \c_msg_coding_error_text_tl
8872 LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
8873 this~property~is~not~defined.
8874 }
8875 \__kernel_msg_new:nnnn { kernel } { scanmark-already-defined }
8876 { Scan~mark~#1~already~defined. }
8877 {
8878 \c_msg_coding_error_text_tl
8879 LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
8880 but~this~name~has~already~been~used~for~a~scan~mark.
8881 }
8882 \__kernel_msg_new:nnnn { kernel } { variable-not-defined }
8883 { Variable~#1~undefined. }
8884 {
8885 \c_msg_coding_error_text_tl
8886 LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
8887 been~defined~yet.

```

```

8888 }
8889 \__kernel_msg_new:nnnn { kernel } { variant-too-long }
8890 { Variant~form~'~#1'~longer~than~base~signature~of~'~#2'. }
8891 {
8892   \c_msg_coding_error_text_tl
8893   LaTeX~has~been~asked~to~create~a~variant~of~the~function~'~#2'~
8894   with~a~signature~starting~with~'~#1',~but~that~is~longer~than~
8895   the~signature~(part~after~the~colon)~of~'~#2'.
8896 }
8897 \__kernel_msg_new:nnnn { kernel } { invalid-variant }
8898 { Variant~form~'~#1'~invalid~for~base~form~'~#2'. }
8899 {
8900   \c_msg_coding_error_text_tl
8901   LaTeX~has~been~asked~to~create~a~variant~of~the~function~'~#2'~
8902   with~a~signature~starting~with~'~#1',~but~cannot~change~an~argument~
8903   from~type~'~#3'~to~type~'~#4'.
8904 }
8905 \__kernel_msg_new:nnnn { kernel } { invalid-exp-args }
8906 { Invalid~variant~specifier~'~#1'~in~'~#2'. }
8907 {
8908   \c_msg_coding_error_text_tl
8909   LaTeX~has~been~asked~to~create~an~\iow_char:N\exp_args:N...~
8910   function~with~signature~'~N#2'~but~'~#1'~is~not~a~valid~argument~
8911   specifier.
8912 }
8913 \__kernel_msg_new:nnn { kernel } { deprecated-variant }
8914 {
8915   Variant~form~'~#1'~deprecated~for~base~form~'~#2'.~
8916   One~should~not~change~an~argument~from~type~'~#3'~to~type~'~#4'
8917   \str_case:nnF {#3}
8918   {
8919     { n } { :~use~a~'\token_if_eq_charcode:NNTF #4 c v V'~variant? }
8920     { N } { :~base~form~only~accepts~a~single~token~argument. }
8921     {#4} { :~base~form~is~already~a~variant. }
8922   } { . }
8923 }

```

Some errors are only needed in package mode if debugging is enabled by one of the options `enable-debug`, `check-declarations`, `log-functions`, or on the contrary if debugging is turned off. In format mode the error is somewhat different.

```

8924 \*package>
8925 \__kernel_if_debug:TF
8926 {
8927   \__kernel_msg_new:nnnn { kernel } { debug }
8928   { The~debugging~option~'~#1'~does~not~exist~\msg_line_context:. }
8929   {
8930     The~functions~'\iow_char:N\debug_on:n'~and~
8931     '\iow_char:N\debug_off:n'~only~accept~the~arguments~
8932     'check-declarations',~'deprecation',~'log-functions',~not~'~#1'.
8933   }
8934   \__kernel_msg_new:nnn { kernel } { expr } { '~#2'~in~#1 }
8935   \__kernel_msg_new:nnnn { kernel } { local-global }
8936   { Inconsistent~local/global~assignment }
8937   {

```

```

8938 \c__msg_coding_error_text_tl
8939 \if:w l #2 Local \else: Global \fi: \ %
8940 assignment~to~a~
8941 \if:w l #1 local
8942 \else:
8943 \if:w g #1 global \else: constant \fi:
8944 \fi:
8945 \ %
8946 variable~'#3'.
8947 }
8948 \__kernel_msg_new:nnnn { kernel } { non-declared-variable }
8949 { The~variable~#1~has~not~been~declared~\msg_line_context:. }
8950 {
8951 \c__msg_coding_error_text_tl
8952 Checking~is~active,~and~you~have~tried~do~so~something~like: \\
8953 \\ \tl_set:Nn ~ #1 ~ \{ ~ ... ~ \} \\
8954 without~first~having: \\
8955 \\ \tl_new:N ~ #1 \\
8956 \\
8957 LaTeX~will~create~the~variable~and~continue.
8958 }
8959 }
8960 {
8961 \__kernel_msg_new:nnnn { kernel } { enable-debug }
8962 { To~use~'#1'~load~expl3~with~the~'enable-debug'~option. }
8963 {
8964 The~function~'#1'~will~be~ignored~because~it~can~only~work~if~
8965 some~internal~functions~in~expl3~have~been~appropriately~
8966 defined.~This~only~happens~if~one~of~the~options~
8967 'enable-debug',~'check-declarations'~or~'log-functions'~was~
8968 given~when~loading~expl3.
8969 }
8970 }
8971 \</package>
8972 \<*/initex>
8973 \__kernel_msg_new:nnnn { kernel } { enable-debug }
8974 { '#1'~cannot~be~used~in~format~mode. }
8975 {
8976 The~function~'#1'~will~be~ignored~because~it~can~only~work~if~
8977 some~internal~functions~in~expl3~have~been~appropriately~
8978 defined.~This~only~happens~in~package~mode~(and~only~if~one~of~
8979 the~options~'enable-debug',~'check-declarations'~or~'log-functions'~
8980 was~given~when~loading~expl3.
8981 }
8982 \</initex>

```

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```

8983 \__kernel_msg_new:nnn { kernel } { bad-exp-end-f }
8984 { Misused~\exp_end_continue_f:w or~:nw }
8985 \__kernel_msg_new:nnn { kernel } { bad-variable }
8986 { Erroneous~variable~#1 used! }
8987 \__kernel_msg_new:nnn { kernel } { misused-sequence }
8988 { A~sequence~was~misused. }

```

```

8989 \__kernel_msg_new:nnn { kernel } { misused-prop }
8990 { A~property~list~was~misused. }
8991 \__kernel_msg_new:nnn { kernel } { negative-replication }
8992 { Negative~argument~for~\prg_replicate:nn. }
8993 \__kernel_msg_new:nnn { kernel } { prop-keyval }
8994 { Missing/extra~'='~in~'#1'~(in~'..._keyval:Nn') }
8995 \__kernel_msg_new:nnn { kernel } { unknown-comparison }
8996 { Relation~'#1'~unknown:~use~=',~<,~>,~==,~!=,~<=,~>=. }
8997 \__kernel_msg_new:nnn { kernel } { zero-step }
8998 { Zero~step~size~for~step~function~#1. }

```

Messages used by the “show” functions.

```

8999 \__kernel_msg_new:nnn { kernel } { show-clist }
9000 {
9001   The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
9002   \tl_if_empty:nTF {#2}
9003   { is~empty \>>~ . }
9004   { contains~the~items~(without~outer~braces): #2 . }
9005 }
9006 \__kernel_msg_new:nnn { kernel } { show-prop }
9007 {
9008   The~property~list~#1~
9009   \tl_if_empty:nTF {#2}
9010   { is~empty \>>~ . }
9011   { contains~the~pairs~(without~outer~braces): #2 . }
9012 }
9013 \__kernel_msg_new:nnn { kernel } { show-seq }
9014 {
9015   The~sequence~#1~
9016   \tl_if_empty:nTF {#2}
9017   { is~empty \>>~ . }
9018   { contains~the~items~(without~outer~braces): #2 . }
9019 }
9020 \__kernel_msg_new:nnn { kernel } { show-streams }
9021 {
9022   \tl_if_empty:nTF {#2} { No~ } { The~following~
9023   \str_case:nn {#1}
9024   {
9025     { ior } { input ~ }
9026     { iow } { output ~ }
9027   }
9028   streams~are~
9029   \tl_if_empty:nTF {#2} { open } { in~use: #2 . }
9030 }

```

17.6 Expandable errors

`__msg_expandable_error:n` In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed \TeX an undefined control sequence, `\LaTeX3 error:`. It is thus interrupted, and shows the context, which thanks to the odd-looking `\use:n` is

```

<argument> \LaTeX3 error:
                The error message.

```

In other words, TeX is processing the argument of `\use:n`, which is `\LaTeX3 error: <error message>`. Then `__msg_expandable_error:w` cleans up. In fact, there is an extra subtlety: if the user inserts tokens for error recovery, they should be kept. Thus we also use an odd space character (with category code 7) and keep tokens until that space character, dropping everything else until `\q_stop`. The `\exp_end:` prevents losing braces around the user-inserted text if any, and stops the expansion of `\exp:w`. The group is used to prevent `\LaTeX3~error:` from being globally equal to `\scan_stop:`.

```

9031 \group_begin:
9032 \cs_set_protected:Npn \__msg_tmp:w #1#2
9033 {
9034   \cs_new:Npn \__msg_expandable_error:n ##1
9035   {
9036     \exp:w
9037     \exp_after:wN \exp_after:wN
9038     \exp_after:wN \__msg_expandable_error:w
9039     \exp_after:wN \exp_after:wN
9040     \exp_after:wN \exp_end:
9041     \use:n { #1 #2 ##1 } #2
9042   }
9043   \cs_new:Npn \__msg_expandable_error:w ##1 #2 ##2 #2 {##1}
9044 }
9045 \exp_args:Ncx \__msg_tmp:w { LaTeX3~error: }
9046 { \char_generate:nn { ' \ } { 7 } }
9047 \group_end:

```

(End definition for `__msg_expandable_error:n` and `__msg_expandable_error:w`.)

```

\__kernel_msg_expandable_error:nnnnnn
\__kernel_msg_expandable_error:nnffff
\__kernel_msg_expandable_error:nnnnn
\__kernel_msg_expandable_error:nnfff
\__kernel_msg_expandable_error:nnnn
\__kernel_msg_expandable_error:nnff
\__kernel_msg_expandable_error:nnn
\__kernel_msg_expandable_error:nnf
\__kernel_msg_expandable_error:nn

```

The command built from the csname `\c__msg_text_prefix_tl LaTeX / #1 / #2` takes four arguments and builds the error text, which is fed to `__msg_expandable_error:n`.

```

9048 \cs_new:Npn \__kernel_msg_expandable_error:nnnnnn #1#2#3#4#5#6
9049 {
9050   \exp_args:Nf \__msg_expandable_error:n
9051   {
9052     \exp_args:NNc \exp_after:wN \exp_stop_f:
9053     { \c__msg_text_prefix_tl LaTeX / #1 / #2 }
9054     {#3} {#4} {#5} {#6}
9055   }
9056 }
9057 \cs_new:Npn \__kernel_msg_expandable_error:nnnnn #1#2#3#4#5
9058 {
9059   \__kernel_msg_expandable_error:nnnnnn
9060   {#1} {#2} {#3} {#4} {#5} { }
9061 }
9062 \cs_new:Npn \__kernel_msg_expandable_error:nnnn #1#2#3#4
9063 {
9064   \__kernel_msg_expandable_error:nnnnnn
9065   {#1} {#2} {#3} {#4} { } { }
9066 }
9067 \cs_new:Npn \__kernel_msg_expandable_error:nnn #1#2#3
9068 {
9069   \__kernel_msg_expandable_error:nnnnnn
9070   {#1} {#2} {#3} { } { } { }
9071 }

```

```

9072 \cs_new:Npn \__kernel_msg_expandable_error:nn #1#2
9073 {
9074   \__kernel_msg_expandable_error:nnnnnn
9075   {#1} {#2} { } { } { } { }
9076 }
9077 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnnnn { nnffff }
9078 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnnn { nnfff }
9079 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnn { nnff }
9080 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnn { nnf }

(End definition for \__kernel_msg_expandable_error:nnnnnn and others.)

9081 </initex | package>

```

18 l3file implementation

The following test files are used for this code: *m3file001*.

```

9082 <*initex | package>

```

18.1 Input operations

```

9083 <@@=ior>

```

18.1.1 Variables and constants

\c_term_ior Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

```

9084 \int_const:Nn \c_term_ior { 16 }

```

(End definition for \c_term_ior. This variable is documented on page 148.)

\g__ior_streams_seq A list of the currently-available input streams to be used as a stack. In format mode, all streams (from 0 to 15) are available, while the package requests streams to L^AT_EX 2_ε as they are needed (initially none are needed), so the starting point varies!

```

9085 \seq_new:N \g__ior_streams_seq
9086 <*initex>
9087 \seq_gset_split:Nnn \g__ior_streams_seq { , }
9088 { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }
9089 </initex>

```

(End definition for \g__ior_streams_seq.)

\l__ior_stream_tl Used to recover the raw stream number from the stack.

```

9090 \tl_new:N \l__ior_stream_tl

```

(End definition for \l__ior_stream_tl.)

\g__ior_streams_prop The name of the file attached to each stream is tracked in a property list. To get the correct number of reserved streams in package mode the underlying mechanism needs to be queried. For L^AT_EX 2_ε and plain T_EX this data is stored in \count16: with the etex package loaded we need to subtract 1 as the register holds the number of the next stream to use. In ConT_EXt, we need to look at \count38 but there is no subtraction: like the original plain T_EX/L^AT_EX 2_ε mechanism it holds the value of the *last* stream allocated.

```

9091 \prop_new:N \g__ior_streams_prop

```

```

9092 \begin{package}
9093 \int_step_inline:nnn
9094 { 0 }
9095 {
9096   \cs_if_exist:NTF \normalend
9097   { \tex_count:D 38 ~ }
9098   {
9099     \tex_count:D 16 ~ %
9100     \cs_if_exist:NT \loccount { - 1 }
9101   }
9102 }
9103 {
9104   \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved-by-format }
9105 }
9106 \end{package}

```

(End definition for \g__ior_streams_prop.)

18.1.2 Stream management

\ior_new:N Reserving a new stream is done by defining the name as equal to using the terminal.

```

\ior_new:c
9107 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }
9108 \cs_generate_variant:Nn \ior_new:N { c }

```

(End definition for \ior_new:N. This function is documented on page 141.)

\g_tmpa_ior The usual scratch space.

```

\g_tmpb_ior
9109 \ior_new:N \g_tmpa_ior
9110 \ior_new:N \g_tmpb_ior

```

(End definition for \g_tmpa_ior and \g_tmpb_ior. These variables are documented on page 148.)

\ior_open:Nn Use the conditional version, with an error if the file is not found.

```

\ior_open:cn
9111 \cs_new_protected:Npn \ior_open:Nn #1#2
9112 { \ior_open:NnF #1 {#2} { \__kernel_file_missing:n {#2} } }
9113 \cs_generate_variant:Nn \ior_open:Nn { c }

```

(End definition for \ior_open:Nn. This function is documented on page 141.)

\l__ior_file_name_str Data storage.

```

9114 \str_new:N \l__ior_file_name_str

```

(End definition for \l__ior_file_name_str.)

\ior_open:NnTF An auxiliary searches for the file in the T_EX, L^AT_EX 2_ε and L^AT_EX 3 paths. Then pass the file found to the lower-level function which deals with streams. The full_name is empty when the file is not found.

```

\ior_open:cnTF
9115 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
9116 {
9117   \file_get_full_name:nN {#2} \l__ior_file_name_str
9118   \str_if_empty:NTF \l__ior_file_name_str
9119   { \prg_return_false: }
9120   {
9121     \__kernel_ior_open:No #1 \l__ior_file_name_str
9122     \prg_return_true:

```



```

9123     }
9124   }
9125   \prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }

```

(End definition for \ior_open:NnTF. This function is documented on page 142.)

__ior_new:N In package mode, streams are reserved using `\newread` before they can be managed by `ior`. To prevent `ior` from being affected by redefinitions of `\newread` (such as done by the third-party package `morewrites`), this macro is saved here under a private name. The complicated code ensures that `__ior_new:N` is not `\outer` despite plain TeX's `\newread` being `\outer`.

```

9126 <*package>
9127 \exp_args:NNf \cs_new_protected:Npn \__ior_new:N
9128   { \exp_args:Nnc \exp_after:wN \exp_stop_f: { newread } }
9129 </package>

```

(End definition for __ior_new:N.)

__kernel_ior_open:Nn The stream allocation itself uses the fact that there is a list of all of those available, so allocation is simply a question of using the number at the top of the list. In package mode, life gets more complex as it's important to keep things in sync. That is done using a two-part approach: any streams that have already been taken up by `ior` but are now free are tracked, so we first try those. If that fails, ask plain TeX or L^AT_EX 2_ε for a new stream and use that number (after a bit of conversion).

```

9130 \cs_new_protected:Npn \__kernel_ior_open:Nn #1#2
9131   {
9132     \ior_close:N #1
9133     \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
9134     { \__ior_open_stream:Nn #1 {#2} }
9135   <*initex>
9136     { \__kernel_msg_fatal:nn { kernel } { input-streams-exhausted } }
9137   </initex>
9138   <*package>
9139     {
9140       \__ior_new:N #1
9141       \tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
9142       \__ior_open_stream:Nn #1 {#2}
9143     }
9144   </package>
9145   }
9146   \cs_generate_variant:Nn \__kernel_ior_open:Nn { No }
9147   \cs_new_protected:Npn \__ior_open_stream:Nn #1#2
9148     {
9149       \tex_global:D \tex_chardef:D #1 = \l__ior_stream_tl \scan_stop:
9150       \prop_gput:NVn \g__ior_streams_prop #1 {#2}
9151       \tex_openin:D #1 #2 \scan_stop:
9152     }

```

(End definition for __kernel_ior_open:Nn and __ior_open_stream:Nn.)

\ior_close:N Closing a stream means getting rid of it at the TeX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range [0, 15]), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

\ior_close:c

```

9153 \cs_new_protected:Npn \ior_close:N #1
9154 {
9155   \int_compare:nT { -1 < #1 < \c_term_ior }
9156   {
9157     \tex_closein:D #1
9158     \prop_gremove:NV \g__ior_streams_prop #1
9159     \seq_if_in:NVF \g__ior_streams_seq #1
9160     { \seq_gpush:NV \g__ior_streams_seq #1 }
9161     \cs_gset_eq:NN #1 \c_term_ior
9162   }
9163 }
9164 \cs_generate_variant:Nn \ior_close:N { c }

```

(End definition for `\ior_close:N`. This function is documented on page 142.)

`\ior_show_list:` Show the property lists, but with some “pretty printing”. See the `l3msg` module. The
`\ior_log_list:` first argument of the message is `ior` (as opposed to `iow`) and the second is empty if no
`__ior_list:N` read stream is open and non-empty (the list of streams formatted using `\msg_show_item_unbraced:nn`) otherwise. The code of the message `show-streams` takes care of translating `ior/iow` to English.

```

9165 \cs_new_protected:Npn \ior_show_list: { \__ior_list:N \msg_show:nnxxxx }
9166 \cs_new_protected:Npn \ior_log_list: { \__ior_list:N \msg_log:nnxxxx }
9167 \cs_new_protected:Npn \__ior_list:N #1
9168 {
9169   #1 { LaTeX / kernel } { show-streams }
9170   { ior }
9171   {
9172     \prop_map_function:NN \g__ior_streams_prop
9173     \msg_show_item_unbraced:nn
9174   }
9175   { } { }
9176 }

```

(End definition for `\ior_show_list:`, `\ior_log_list:`, and `__ior_list:N`. These functions are documented on page 142.)

18.1.3 Reading input

`\if_eof:w` The primitive conditional

```

9177 \cs_new_eq:NN \if_eof:w \tex_ifeof:D

```

(End definition for `\if_eof:w`. This function is documented on page 148.)

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.
`\ior_if_eof:NTF` The primitive test can only deal with numbers in the range $[0, 15]$ so we catch outliers (they are exhausted).

```

9178 \prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
9179 {
9180   \cs_if_exist:NTF #1
9181   {
9182     \int_compare:nTF { -1 < #1 < \c_term_ior }
9183     {
9184       \if_eof:w #1
9185       \prg_return_true:

```

```

9186         \else:
9187         \prg_return_false:
9188         \fi:
9189     }
9190     { \prg_return_true: }
9191 }
9192 { \prg_return_true: }
9193 }

```

(End definition for `\ior_if_eof:NTF`. This function is documented on page 145.)

\ior_get:NN And here we read from files.

```

9194 \cs_new_protected:Npn \ior_get:NN #1#2
9195 { \tex_read:D #1 to #2 }

```

(End definition for `\ior_get:NN`. This function is documented on page 143.)

\ior_str_get:NN Reading as strings is a more complicated wrapper, as we wish to remove the endline character and restore it afterwards.

```

9196 \cs_new_protected:Npn \ior_str_get:NN #1#2
9197 {
9198     \exp_args:Nno \use:n
9199     {
9200         \int_set:Nn \tex_endlinechar:D { -1 }
9201         \etex_readline:D #1 to #2
9202         \int_set:Nn \tex_endlinechar:D
9203     } { \int_use:N \tex_endlinechar:D }
9204 }

```

(End definition for `\ior_str_get:NN`. This function is documented on page 143.)

\ior_map_break: Usual map breaking functions.

```

\ior_map_break:n
9205 \cs_new:Npn \ior_map_break:
9206 { \prg_map_break:Nn \ior_map_break: { } }
9207 \cs_new:Npn \ior_map_break:n
9208 { \prg_map_break:Nn \ior_map_break: }

```

(End definition for `\ior_map_break:` and `\ior_map_break:n`. These functions are documented on page 144.)

\ior_map_inline:Nn Mapping to an input stream can be done on either a token or a string basis, hence the
\ior_str_map_inline:Nn set up. Within that, there is a check to avoid reading past the end of a file, hence the
__ior_map_inline:NNn two applications of `\ior_if_eof:N`. This mapping cannot be nested with twice the same
__ior_map_inline:NNNn stream, as the stream has only one “current line”.
__ior_map_inline_loop:NNN
\l__ior_internal_tl

```

9209 \cs_new_protected:Npn \ior_map_inline:Nn
9210 { \__ior_map_inline:NNn \ior_get:NN }
9211 \cs_new_protected:Npn \ior_str_map_inline:Nn
9212 { \__ior_map_inline:NNn \ior_str_get:NN }
9213 \cs_new_protected:Npn \__ior_map_inline:NNn
9214 {
9215     \int_gincr:N \g__kernel_prg_map_int
9216     \exp_args:Nc \__ior_map_inline:NNNn
9217     { \__ior_map_ \int_use:N \g__kernel_prg_map_int :n }
9218 }
9219 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4

```

```

9220 {
9221   \cs_gset_protected:Npn #1 ##1 {#4}
9222   \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
9223   \prg_break_point:Nn \ior_map_break:
9224     { \int_gdecr:N \g__kernel_prg_map_int }
9225 }
9226 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
9227 {
9228   #2 #3 \l__ior_internal_tl
9229   \ior_if_eof:NF #3
9230   {
9231     \exp_args:No #1 \l__ior_internal_tl
9232     \__ior_map_inline_loop:NNN #1#2#3
9233   }
9234 }
9235 \tl_new:N \l__ior_internal_tl

```

(End definition for `\ior_map_inline:Nn` and others. These functions are documented on page 144.)

18.2 Output operations

```

9236 <@@=iow>

```

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

18.2.1 Variables and constants

`\c_log_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`)
`\c_term_iow` and to both the terminal and transcript file (`\c_term_iow`). Recent LuaTeX provide 128 write streams; we also use `\c_term_iow` as the first non-allowed write stream so its value depends on the engine.

```

9237 \int_const:Nn \c_log_iow { -1 }
9238 \int_const:Nn \c_term_iow
9239 {
9240   \bool_lazy_and:nnTF
9241     { \sys_if_engine luatex_p: }
9242     { \int_compare_p:nNn \luatex luatexversion:D > { 80 } }
9243     { 128 }
9244     { 16 }
9245 }

```

(End definition for `\c_log_iow` and `\c_term_iow`. These variables are documented on page 148.)

`\g__iow_streams_seq` A list of the currently-available output streams to be used as a stack. The stream 18 is special, as `\write18` is used to denote commands to be sent to the OS.

```

9246 \seq_new:N \g__iow_streams_seq
9247 (*initex)
9248 \exp_args:Nnx \use:n
9249 { \seq_gset_split:Nnn \g__iow_streams_seq { } }
9250 {
9251   \int_step_function:nnN { 0 } { \c_term_iow }
9252   \prg_do_nothing:
9253 }
9254 \int_compare:nNnF \c_term_iow < { 18 }

```

```

9255 { \seq_gremove_all:Nn \g__iow_streams_seq { 18 } }
9256 \</initex>

```

(End definition for \g__iow_streams_seq.)

\l__iow_stream_tl Used to recover the raw stream number from the stack.

```

9257 \tl_new:N \l__iow_stream_tl

```

(End definition for \l__iow_stream_tl.)

\g__iow_streams_prop As for reads with the appropriate adjustment of the register numbers to check on.

```

9258 \prop_new:N \g__iow_streams_prop
9259 \*package>
9260 \int_step_inline:nnn
9261 { 0 }
9262 {
9263   \cs_if_exist:NTF \normalend
9264   { \tex_count:D 39 ~ }
9265   {
9266     \tex_count:D 17 ~
9267     \cs_if_exist:NT \loccount { - 1 }
9268   }
9269 }
9270 {
9271   \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved-by~format }
9272 }
9273 \</package>

```

(End definition for \g__iow_streams_prop.)

18.3 Stream management

\iow_new:N Reserving a new stream is done by defining the name as equal to writing to the terminal:
\iow_new:c odd but at least consistent.

```

9274 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
9275 \cs_generate_variant:Nn \iow_new:N { c }

```

(End definition for \iow_new:N. This function is documented on page 141.)

\g_tmpa_iow The usual scratch space.

\g_tmpb_iow

```

9276 \iow_new:N \g_tmpa_iow
9277 \iow_new:N \g_tmpb_iow

```

(End definition for \g_tmpa_iow and \g_tmpb_iow. These variables are documented on page 148.)

__iow_new:N As for read streams, copy \newwrite in package mode, making sure that it is not \outer.

```

9278 \*package>
9279 \exp_args:NNf \cs_new_protected:Npn \__iow_new:N
9280 { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }
9281 \</package>

```

(End definition for __iow_new:N.)

\l__iow_file_name_str Data storage.

```

9282 \str_new:N \l__iow_file_name_str

```

(End definition for \l__iow_file_name_str.)

\iow_open:Nn The same idea as for reading, but without the path and without the need to allow for a conditional version.
\iow_open:cn

```

\__iow_open_stream:Nn 9283 \cs_new_protected:Npn \iow_open:Nn #1#2
\__iow_open_stream:NV 9284 {
9285   \__kernel_file_name_sanitiz:n {#2} \l__iow_file_name_str
9286   \iow_close:N #1
9287   \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
9288   { \__iow_open_stream:NV #1 \l__iow_file_name_str }
9289   \*initex
9290   { \__kernel_msg_fatal:nn { kernel } { output-streams-exhausted } }
9291   \*initex
9292   \*package
9293   {
9294     \__iow_new:N #1
9295     \tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
9296     \__iow_open_stream:NV #1 \l__iow_file_name_str
9297   }
9298   \*package
9299 }
9300 \cs_generate_variant:Nn \iow_open:Nn { c }
9301 \cs_new_protected:Npn \__iow_open_stream:Nn #1#2
9302 {
9303   \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
9304   \prop_gput:Nv \g__iow_streams_prop #1 {#2}
9305   \tex_immediate:D \tex_openout:D #1 #2 \scan_stop:
9306 }
9307 \cs_generate_variant:Nn \__iow_open_stream:Nn { NV }

```

(End definition for \iow_open:Nn and __iow_open_stream:Nn. This function is documented on page 142.)

\iow_close:N Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.
\iow_close:c

```

9308 \cs_new_protected:Npn \iow_close:N #1
9309 {
9310   \int_compare:nT { - \c_log_iow < #1 < \c_term_iow }
9311   {
9312     \tex_immediate:D \tex_closeout:D #1
9313     \prop_gremove:Nv \g__iow_streams_prop #1
9314     \seq_if_in:NvF \g__iow_streams_seq #1
9315     { \seq_gpush:Nv \g__iow_streams_seq #1 }
9316     \cs_gset_eq:NN #1 \c_term_iow
9317   }
9318 }
9319 \cs_generate_variant:Nn \iow_close:N { c }

```

(End definition for \iow_close:N. This function is documented on page 142.)

\iow_show_list: Done as for input, but with a copy of the auxiliary so the name is correct.

\iow_log_list: 9320 \cs_new_protected:Npn \iow_show_list: { __iow_list:N \msg_show:nnxxxx }
__iow_list:N 9321 \cs_new_protected:Npn \iow_log_list: { __iow_list:N \msg_log:nnxxxx }

```

9322 \cs_new_protected:Npn \__iow_list:N #1
9323 {
9324     #1 { LaTeX / kernel } { show-streams }
9325     { iow }
9326     {
9327         \prop_map_function:NN \g__iow_streams_prop
9328         \msg_show_item_unbraced:nn
9329     }
9330     { } { }
9331 }

```

(End definition for `\iow_show_list:`, `\iow_log_list:`, and `__iow_list:N`. These functions are documented on page 142.)

18.3.1 Deferred writing

`\iow_shipout_x:Nn` First the easy part, this is the primitive, which expects its argument to be braced.

```

\iow_shipout_x:Nx 9332 \cs_new_protected:Npn \iow_shipout_x:Nn #1#2
\iow_shipout_x:cn 9333 { \tex_write:D #1 {#2} }
\iow_shipout_x:cx 9334 \cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }

```

(End definition for `\iow_shipout_x:Nn`. This function is documented on page 146.)

`\iow_shipout:Nn` With ε -TeX available deferred writing without expansion is easy.

```

\iow_shipout:Nx 9335 \cs_new_protected:Npn \iow_shipout:Nn #1#2
\iow_shipout:cn 9336 { \tex_write:D #1 { \exp_not:n {#2} } }
\iow_shipout:cx 9337 \cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }

```

(End definition for `\iow_shipout:Nn`. This function is documented on page 146.)

18.3.2 Immediate writing

`__kernel_iow_with:Nnn` If the integer #1 is equal to #2, just leave #3 in the input stream. Otherwise, pass the old value to an auxiliary, which sets the integer to the new value, runs the code, and restores the integer.

```

9338 \cs_new_protected:Npn \__kernel_iow_with:Nnn #1#2
9339 {
9340     \int_compare:nNnTF {#1} = {#2}
9341     { \use:n }
9342     { \exp_args:No \__iow_with:nNnn { \int_use:N #1 } #1 {#2} }
9343 }
9344 \cs_new_protected:Npn \__iow_with:nNnn #1#2#3#4
9345 {
9346     \int_set:Nn #2 {#3}
9347     #4
9348     \int_set:Nn #2 {#1}
9349 }

```

(End definition for `__kernel_iow_with:Nnn` and `__iow_with:nNnn`.)

`\iow_now:Nn` This routine writes the second argument onto the output stream without expansion. If
`\iow_now:Nx` this stream isn't open, the output goes to the terminal instead. If the first argument is
`\iow_now:cn` no output stream at all, we get an internal error. We don't use the expansion done by
`\iow_now:cx` `\write` to get the Nx variant, because it differs in subtle ways from x-expansion, namely,

macro parameter characters would not need to be doubled. We set the `\newlinechar` to 10 using `__kernel_iow_with:Nnn` to support formats such as plain T_EX: otherwise, `\iow_newline:` would not work. We do not do this for `\iow_shipout:Nn` or `\iow_shipout_x:Nn`, as T_EX looks at the value of the `\newlinechar` at shipout time in those cases.

```

9350 \cs_new_protected:Npn \iow_now:Nn #1#2
9351 {
9352   \__kernel_iow_with:Nnn \tex_newlinechar:D { ‘\^^J }
9353   { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
9354 }
9355 \cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }

```

(End definition for `\iow_now:Nn`. This function is documented on page 145.)

`\iow_log:n` Writing to the log and the terminal directly are relatively easy.

`\iow_log:x` `9356 \cs_set_protected:Npn \iow_log:x { \iow_now:Nx \c_log_iow }`

`\iow_term:n` `9357 \cs_new_protected:Npn \iow_log:n { \iow_now:Nn \c_log_iow }`

`\iow_term:x` `9358 \cs_set_protected:Npn \iow_term:x { \iow_now:Nx \c_term_iow }`

`9359 \cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }`

(End definition for `\iow_log:n` and `\iow_term:n`. These functions are documented on page 145.)

18.3.3 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written to an output stream.

```

9360 \cs_new:Npn \iow_newline: { ^^J }

```

(End definition for `\iow_newline:`. This function is documented on page 146.)

`\iow_char:N` Function to write any escaped char to an output stream.

```

9361 \cs_new_eq:NN \iow_char:N \cs_to_str:N

```

(End definition for `\iow_char:N`. This function is documented on page 146.)

18.3.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

`\l_iow_line_count_int` This is the “raw” number of characters in a line which can be written to the terminal. The standard value is the line length typically used by T_EXLive and MiK_TE_X.

```

9362 \int_new:N \l_iow_line_count_int
9363 \int_set:Nn \l_iow_line_count_int { 78 }

```

(End definition for `\l_iow_line_count_int`. This variable is documented on page 147.)

`\l__iow_newline_tl` The token list inserted to produce a new line, with the *⟨run-on text⟩*.

```

9364 \tl_new:N \l__iow_newline_tl

```

(End definition for `\l__iow_newline_tl`.)

`\l_iow_line_target_int` This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.

```

9365 \int_new:N \l_iow_line_target_int

```


(End definition for \l__iow_line_target_int.)

__iow_set_indent:n The one_indent variables hold one indentation marker and its length. The __iow_unindent:w auxiliary removes one indentation. The function __iow_set_indent:n (that could possibly be public) sets the indentation in a consistent way. We set it to four spaces by default.

```

9366 \tl_new:N \l__iow_one_indent_tl
9367 \int_new:N \l__iow_one_indent_int
9368 \cs_new:Npn \__iow_unindent:w { }
9369 \cs_new_protected:Npn \__iow_set_indent:n #1
9370 {
9371   \tl_set:Nx \l__iow_one_indent_tl
9372     { \exp_args:No \__kernel_str_to_other_fast:n { \tl_to_str:n {#1} } }
9373   \int_set:Nn \l__iow_one_indent_int
9374     { \str_count:N \l__iow_one_indent_tl }
9375   \exp_last_unbraced:NNo
9376     \cs_set:Npn \__iow_unindent:w \l__iow_one_indent_tl { }
9377 }
9378 \exp_args:Nx \__iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } }

```

(End definition for __iow_set_indent:n and others.)

\l__iow_indent_tl The current indentation (some copies of \l__iow_one_indent_tl) and its number of characters.

```

9379 \tl_new:N \l__iow_indent_tl
9380 \int_new:N \l__iow_indent_int

```

(End definition for \l__iow_indent_tl and \l__iow_indent_int.)

\l__iow_line_tl These hold the current line of text and a partial line to be added to it, respectively.

```

9381 \tl_new:N \l__iow_line_tl
9382 \tl_new:N \l__iow_line_part_tl

```

(End definition for \l__iow_line_tl and \l__iow_line_part_tl.)

\l__iow_line_break_bool Indicates whether the line was broken precisely at a chunk boundary.

```

9383 \bool_new:N \l__iow_line_break_bool

```

(End definition for \l__iow_line_break_bool.)

\l__iow_wrap_tl Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

```

9384 \tl_new:N \l__iow_wrap_tl

```

(End definition for \l__iow_wrap_tl.)

\c__iow_wrap_marker_tl Every special action of the wrapping code is starts with the same recognizable string, \c__iow_wrap_end_marker_tl \c__iow_wrap_marker_tl. Upon seeing that “word”, the wrapping code reads one space-delimited argument to know what operation to perform. The setting of \escapechar here is not very important, but makes \c__iow_wrap_marker_tl look marginally nicer.

```

9385 \group_begin:
9386   \int_set:Nn \tex_escapechar:D { -1 }
9387   \tl_const:Nx \c__iow_wrap_marker_tl
9388     { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }

```

```

9389 \group_end:
9390 \tl_map_inline:nn
9391 { { end } { newline } { indent } { unindent } }
9392 {
9393   \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
9394   {
9395     \c__iow_wrap_marker_tl
9396     #1
9397     \c_catcode_other_space_tl
9398   }
9399 }

```

(End definition for `\c__iow_wrap_marker_tl` and others.)

`\iow_indent:n` We set `\iow_indent:n` to produce an error when outside messages. Within wrapped message, it is set to `__iow_indent:n` when valid and otherwise to `__iow_indent_error:n`.
`__iow_indent:n` The first places the instruction for increasing the indentation before its argument, and
`__iow_indent_error:n` the instruction for unindenting afterwards. The second produces an error expandably. Note that there are no forced line-break, so the indentation only changes when the next line is started.

```

9400 \cs_new_protected:Npn \iow_indent:n #1
9401 {
9402   \__kernel_msg_error:nnnnn { kernel } { iow-indent }
9403   { \iow_wrap:nnnN } { \iow_indent:n } {#1}
9404   #1
9405 }
9406 \cs_new:Npx \__iow_indent:n #1
9407 {
9408   \c__iow_wrap_indent_marker_tl
9409   #1
9410   \c__iow_wrap_unindent_marker_tl
9411 }
9412 \cs_new:Npn \__iow_indent_error:n #1
9413 {
9414   \__kernel_msg_expandable_error:nnnnn { kernel } { iow-indent }
9415   { \iow_wrap:nnnN } { \iow_indent:n } {#1}
9416   #1
9417 }

```

(End definition for `\iow_indent:n`, `__iow_indent:n`, and `__iow_indent_error:n`. This function is documented on page 147.)

`\iow_wrap:nnnN` The main wrapping function works as follows. First give `\`, `_` and other formatting commands the correct definition for messages and perform the given setup #3. The definition of `_` uses an “other” space rather than a normal space, because the latter might be absorbed by \TeX to end a number or other f-type expansions. Use `\conditionally@traceoff` if defined; it is introduced by the `trace` package and suppresses uninteresting tracing of the wrapping code.

```

9418 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
9419 {
9420   \group_begin:
9421   (package) \use:c { conditionally@traceoff }
9422   \int_set:Nn \tex_escapechar:D { -1 }
9423   \cs_set:Npx \{ { \token_to_str:N \{ }

```

```

9424 \cs_set:Npx \# { \token_to_str:N \# }
9425 \cs_set:Npx \} { \token_to_str:N \} }
9426 \cs_set:Npx \% { \token_to_str:N \% }
9427 \cs_set:Npx \~ { \token_to_str:N \~ }
9428 \int_set:Nn \tex_escapechar:D { 92 }
9429 \cs_set_eq:NN \\\ \iow_newline:
9430 \cs_set_eq:NN \ \c_catcode_other_space_tl
9431 \cs_set_eq:NN \iow_indent:n \__iow_indent:n
9432 #3

```

Then fully-expand the input: in package mode, the expansion uses L^AT_EX 2_ε's `\protect` mechanism in the same way as `\typeout`. In generic mode this setting is useless but harmless. As soon as the expansion is done, reset `\iow_indent:n` to its error definition: it only works in the first argument of `\iow_wrap:nnnN`.

```

9433 <package> \cs_set_eq:NN \protect \token_to_str:N
9434 \tl_set:Nx \l__iow_wrap_tl {#1}
9435 \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n

```

Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and initialize the target count for lines (the first line has target count `\l__iow_line_count_int` instead).

```

9436 \tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
9437 \tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
9438 \int_set:Nn \l__iow_line_target_int
9439 { \l__iow_line_count_int - \str_count:N \l__iow_newline_tl + 1 }

```

There is then a loop over the input, which stores the wrapped result in `\l__iow_wrap_tl`. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The `\tl_to_str:N` step converts the “other” spaces back to normal spaces. The f-expansion removes a leading space from `\l__iow_wrap_tl`.

```

9440 \__iow_wrap_do:
9441 \exp_args:NNf \group_end:
9442 #4 { \tl_to_str:N \l__iow_wrap_tl }
9443 }

```

(End definition for `\iow_wrap:nnnN`. This function is documented on page 147.)

`__iow_wrap_do:` Escape spaces and change newlines to `\c__iow_wrap_newline_marker_tl`. Set up a few variables, in particular the initial value of `\l__iow_wrap_tl`: the space stops the f-expansion of the main wrapping function and `\use_none:n` removes a newline marker inserted by later code. The main loop consists of repeatedly calling the `chunk` auxiliary to wrap chunks delimited by (newline or indentation) markers.

```

9444 \cs_new_protected:Npn \__iow_wrap_do:
9445 {
9446   \tl_set:Nx \l__iow_wrap_tl
9447   {
9448     \exp_args:No \__kernel_str_to_other_fast:n \l__iow_wrap_tl
9449     \c__iow_wrap_end_marker_tl
9450   }
9451   \tl_set:Nx \l__iow_wrap_tl
9452   {
9453     \exp_after:wN \__iow_wrap_fix_newline:w \l__iow_wrap_tl
9454     ^^J \q_nil ^^J \q_stop
9455   }

```

```

9456 \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
9457 }
9458 \cs_new:Npn \__iow_wrap_fix_newline:w #1 ^^J #2 ^^J
9459 {
9460   #1
9461   \if_meaning:w \q_nil #2
9462     \use_i_delimit_by_q_stop:nw
9463   \fi:
9464   \c__iow_wrap_newline_marker_tl
9465   \__iow_wrap_fix_newline:w #2 ^^J
9466 }
9467 \cs_new_protected:Npn \__iow_wrap_start:w
9468 {
9469   \bool_set_false:N \l__iow_line_break_bool
9470   \tl_clear:N \l__iow_line_tl
9471   \tl_clear:N \l__iow_line_part_tl
9472   \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
9473   \int_zero:N \l__iow_indent_int
9474   \tl_clear:N \l__iow_indent_tl
9475   \__iow_wrap_chunk:nw { \l__iow_line_count_int }
9476 }

```

(End definition for __iow_wrap_do:, __iow_wrap_fix_newline:w, and __iow_wrap_start:w.)

__iow_wrap_chunk:nw The **chunk** and **next** auxiliaries are defined indirectly to obtain the expansions of \c_catcode_other_space_tl and \c__iow_wrap_marker_tl in their definition. The **next** auxiliary calls a function corresponding to the type of marker (its ##2), which can be **newline** or **indent** or **unindent** or **end**. The first argument of the **chunk** auxiliary is a target number of characters and the second is some string to wrap. If the chunk is empty simply call **next**. Otherwise, set up a call to __iow_wrap_line:nw, including the indentation if the current line is empty, and including a trailing space (#1) before the __iow_wrap_end_chunk:w auxiliary.

```

9477 \cs_set_protected:Npn \__iow_tmp:w #1#2
9478 {
9479   \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
9480   {
9481     \tl_if_empty:NTF {##2}
9482     {
9483       \tl_clear:N \l__iow_line_part_tl
9484       \__iow_wrap_next:nw {##1}
9485     }
9486     {
9487       \tl_if_empty:NTF \l__iow_line_tl
9488       {
9489         \__iow_wrap_line:nw
9490         { \l__iow_indent_tl }
9491         ##1 - \l__iow_indent_int ;
9492       }
9493       { \__iow_wrap_line:nw { } ##1 ; }
9494       ##2 #1
9495       \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \q_stop
9496     }
9497   }
9498   \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1

```

```

9499     { \use:c { __iow_wrap_##2:n } {##1} }
9500   }
9501 \exp_args:NVV \__iow_tmp:w \c_catcode_other_space_tl \c__iow_wrap_marker_tl

```

(End definition for __iow_wrap_chunk:nw and __iow_wrap_next:nw.)

<pre> __iow_wrap_line:nw __iow_wrap_line_loop:w __iow_wrap_line_aux:Nw __iow_wrap_line_end:NnnnnnnN __iow_wrap_line_end:nw __iow_wrap_end_chunk:w </pre>	<p>This is followed by $\{\langle string \rangle\} \langle intexpr \rangle$; . It stores the $\langle string \rangle$ and up to $\langle intexpr \rangle$ characters from the current chunk into <code>\l__iow_line_part_tl</code>. Characters are grabbed 8 at a time and left in <code>\l__iow_line_part_tl</code> by the <code>line_loop</code> auxiliary. When $k < 8$ remain to be found, the <code>line_aux</code> auxiliary calls the <code>line_end</code> auxiliary followed by (the single digit) k, then $7 - k$ empty brace groups, then the chunk's remaining characters. The <code>line_end</code> auxiliary leaves k characters from the chunk in the line part, then ends the assignment. Ignore the <code>\use_none:nnnnn</code> line for now. If the next character is a space the line can be broken there: store what we found into the result and get the next line. Otherwise some work is needed to find a break-point. So far we have ignored what happens if the chunk is shorter than the requested number of characters: this is dealt with by the <code>end_chunk</code> auxiliary, which gets treated like a character by the rest of the code. It ends up being called either as one of the arguments #2-#9 of the <code>line_loop</code> auxiliary or as one of the arguments #2-#8 of the <code>line_end</code> auxiliary. In both cases stop the assignment and work out how many characters are still needed. The weird <code>\use_none:nnnnn</code> ensures that the required data is in the right place.</p>
--	--

```

9502 \cs_new_protected:Npn \__iow_wrap_line:nw #1
9503 {
9504   \tex_edef:D \l__iow_line_part_tl { \if_false: } \fi:
9505   #1
9506   \exp_after:wN \__iow_wrap_line_loop:w
9507   \int_value:w \int_eval:w
9508 }
9509 \cs_new:Npn \__iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
9510 {
9511   \if_int_compare:w #1 < 8 \exp_stop_f:
9512     \__iow_wrap_line_aux:Nw #1
9513   \fi:
9514   #2 #3 #4 #5 #6 #7 #8 #9
9515   \exp_after:wN \__iow_wrap_line_loop:w
9516   \int_value:w \int_eval:w #1 - 8 ;
9517 }
9518 \cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
9519 {
9520   #2
9521   \exp_after:wN \__iow_wrap_line_end:NnnnnnnN
9522   \exp_after:wN #1
9523   \exp:w \exp_end_continue_f:w
9524   \exp_after:wN \exp_after:wN
9525   \if_case:w #1 \exp_stop_f:
9526     \prg_do_nothing:
9527   \or: \use_none:n
9528   \or: \use_none:nn
9529   \or: \use_none:nnn
9530   \or: \use_none:nnnn
9531   \or: \use_none:nnnnn
9532   \or: \use_none:nnnnnn
9533   \or: \use_none:nnnnnnn

```

```

9534     \fi:
9535     { } { } { } { } { } { } { } { } { } #3
9536   }
9537 \cs_new:Npn \__iow_wrap_line_end:NnnnnnnnN #1#2#3#4#5#6#7#8#9
9538 {
9539     #2 #3 #4 #5 #6 #7 #8
9540     \use_none:nnnnn \int_eval:w 8 - ; #9
9541     \token_if_eq_charcode:NNTF \c_space_token #9
9542     { \__iow_wrap_line_end:nw { } }
9543     { \if_false: { \fi: } \__iow_wrap_break:w #9 }
9544 }
9545 \cs_new:Npn \__iow_wrap_line_end:nw #1
9546 {
9547     \if_false: { \fi: }
9548     \__iow_wrap_store_do:n {#1}
9549     \__iow_wrap_next_line:w
9550 }
9551 \cs_new:Npn \__iow_wrap_end_chunk:w
9552 #1 \int_eval:w #2 - #3 ; #4#5 \q_stop
9553 {
9554     \if_false: { \fi: }
9555     \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
9556 }

```

(End definition for __iow_wrap_line:nw and others.)

<pre> __iow_wrap_break:w __iow_wrap_break_first:w __iow_wrap_break_none:w __iow_wrap_break_loop:w __iow_wrap_break_end:w </pre>	<p>Functions here are defined indirectly: __iow_tmp:w is eventually called with an “other” space as its argument. The goal is to remove from \l__iow_line_part_tl the part after the last space. In most cases this is done by repeatedly calling the break_loop auxiliary, which leaves “words” (delimited by spaces) until it hits the trailing space: then its argument ##3 is ? __iow_wrap_break_end:w instead of a single token, and that break_end auxiliary leaves in the assignment the line until the last space, then calls __iow_wrap_line_end:nw to finish up the line and move on to the next. If there is no space in \l__iow_line_part_tl then the break_first auxiliary calls the break_none auxiliary. In that case, if the current line is empty, the complete word (including ##4, characters beyond what we had grabbed) is added to the line, making it over-long. Otherwise, the word is used for the following line (and the last space of the line so far is removed because it was inserted due to the presence of a marker).</p>
--	---

```

9557 \cs_set_protected:Npn \__iow_tmp:w #1
9558 {
9559     \cs_new:Npn \__iow_wrap_break:w
9560     {
9561         \tex_edef:D \l__iow_line_part_tl
9562         { \if_false: } \fi:
9563         \exp_after:wN \__iow_wrap_break_first:w
9564         \l__iow_line_part_tl
9565         #1
9566         { ? \__iow_wrap_break_end:w }
9567         \q_mark
9568     }
9569 \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
9570 {
9571     \use_none:nn ##2 \__iow_wrap_break_none:w

```

```

9572     \_iow_wrap_break_loop:w ##1 #1 ##2
9573   }
9574   \cs_new:Npn \_iow_wrap_break_none:w ##1##2 #1 ##3 \q_mark ##4 #1
9575   {
9576     \tl_if_empty:NTF \l__iow_line_tl
9577     { ##2 ##4 \_iow_wrap_line_end:nw { } }
9578     { \_iow_wrap_line_end:nw { \_iow_wrap_trim:N } ##2 ##4 #1 }
9579   }
9580   \cs_new:Npn \_iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
9581   {
9582     \use_none:n ##3
9583     ##1 #1
9584     \_iow_wrap_break_loop:w ##2 #1 ##3
9585   }
9586   \cs_new:Npn \_iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \q_mark
9587   { ##1 \_iow_wrap_line_end:nw { } ##3 }
9588 }
9589 \exp_args:NV \_iow_tmp:w \c_catcode_other_space_tl

```

(End definition for `_iow_wrap_break:w` and others.)

`_iow_wrap_next_line:w` The special case where the end of a line coincides with the end of a chunk is detected here, to avoid a spurious empty line. Otherwise, call `_iow_wrap_line:nw` to find characters for the next line (remembering to account for the indentation).

```

9590 \cs_new_protected:Npn \_iow_wrap_next_line:w #1#2 \q_stop
9591 {
9592   \tl_clear:N \l__iow_line_tl
9593   \token_if_eq_meaning:NNTF #1 \_iow_wrap_end_chunk:w
9594   {
9595     \tl_clear:N \l__iow_line_part_tl
9596     \bool_set_true:N \l__iow_line_break_bool
9597     \_iow_wrap_next:nw { \l__iow_line_target_int }
9598   }
9599   {
9600     \_iow_wrap_line:nw
9601     { \l__iow_indent_tl }
9602     \l__iow_line_target_int - \l__iow_indent_int ;
9603     #1 #2 \q_stop
9604   }
9605 }

```

(End definition for `_iow_wrap_next_line:w`.)

`_iow_wrap_indent:` These functions are called after a chunk has been wrapped, when encountering **indent/unindent** markers. Add the line part (last line part of the previous chunk) to the line so far and reset a boolean denoting the presence of a line-break. Most importantly, add or remove one indent from the current indent (both the integer and the token list). Finally, continue wrapping.

```

9606 \cs_new_protected:Npn \_iow_wrap_indent:n #1
9607 {
9608   \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
9609   \bool_set_false:N \l__iow_line_break_bool
9610   \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
9611   \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }

```

```

9612     \__iow_wrap_chunk:nw {#1}
9613   }
9614 \cs_new_protected:Npn \__iow_wrap_unindent:n #1
9615   {
9616     \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
9617     \bool_set_false:N \l__iow_line_break_bool
9618     \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
9619     \tl_set:Nx \l__iow_indent_tl
9620       { \exp_after:wN \__iow_unindent:w \l__iow_indent_tl }
9621     \__iow_wrap_chunk:nw {#1}
9622   }

```

(End definition for __iow_wrap_indent: and __iow_wrap_unindent:.)

__iow_wrap_newline: These functions are called after a chunk has been line-wrapped, when encountering a
 __iow_wrap_end: **newline/end** marker. Unless we just took a line-break, store the line part and the line
 so far into the whole \l__iow_wrap_tl, trimming a trailing space. In the **newline** case
 look for a new line (of length \l__iow_line_target_int) in a new chunk.

```

9623 \cs_new_protected:Npn \__iow_wrap_newline:n #1
9624   {
9625     \bool_if:NF \l__iow_line_break_bool
9626       { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
9627     \bool_set_false:N \l__iow_line_break_bool
9628     \__iow_wrap_chunk:nw { \l__iow_line_target_int }
9629   }
9630 \cs_new_protected:Npn \__iow_wrap_end:n #1
9631   {
9632     \bool_if:NF \l__iow_line_break_bool
9633       { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
9634     \bool_set_false:N \l__iow_line_break_bool
9635   }

```

(End definition for __iow_wrap_newline: and __iow_wrap_end:.)

__iow_wrap_store_do:n First add the last line part to the line, then append it to \l__iow_wrap_tl with the
 appropriate new line (with “run-on” text), possibly with its last space removed (#1 is
 empty or __iow_wrap_trim:N).

```

9636 \cs_new_protected:Npn \__iow_wrap_store_do:n #1
9637   {
9638     \tl_set:Nx \l__iow_line_tl
9639       { \l__iow_line_tl \l__iow_line_part_tl }
9640     \tl_set:Nx \l__iow_wrap_tl
9641       {
9642         \l__iow_wrap_tl
9643         \l__iow_newline_tl
9644         #1 \l__iow_line_tl
9645       }
9646     \tl_clear:N \l__iow_line_tl
9647   }

```

(End definition for __iow_wrap_store_do:n.)

__iow_wrap_trim:N Remove one trailing “other” space from the argument.

```

\__iow_wrap_trim:w 9648 \cs_new:Npn \__iow_wrap_trim:N #1

```



```

9649 { \tl_if_empty:NF #1 { \exp_after:wN \__iow_wrap_trim:w #1 \q_stop } }
9650 \exp_last_unbraced:NNNNo
9651 \cs_new:Npn \__iow_wrap_trim:w #1 \c_catcode_other_space_tl \q_stop {#1}

(End definition for \__iow_wrap_trim:N and \__iow_wrap_trim:w.)

9652 <@@=file>

```

18.4 File operations

`\g__file_internal_ior` A reserved stream to test for file existence.

```

9653 \ior_new:N \g__file_internal_ior

(End definition for \g__file_internal_ior.)

```

`\g_file_curr_dir_str` The name of the current file should be available at all times. For the format the file name needs to be picked up at the start of the run. In L^AT_EX 2_ε package mode the current file name is collected from `\@currname`.

```

\g_file_curr_ext_str
\g_file_curr_name_str
9654 \str_new:N \g_file_curr_dir_str
9655 \str_new:N \g_file_curr_ext_str
9656 \str_new:N \g_file_curr_name_str
9657 <*initex>
9658 \tex_everyjob:D \exp_after:wN
9659 {
9660   \tex_the:D \tex_everyjob:D
9661   \str_gset:Nx \g_file_curr_name_str { \tex_jobname:D }
9662 }
9663 </initex>
9664 <*package>
9665 \cs_if_exist:NT \@currname
9666 { \str_gset_eq:NN \g_file_curr_name_str \@currname }
9667 </package>

```

(End definition for `\g_file_curr_dir_str`, `\g_file_curr_ext_str`, and `\g_file_curr_name_str`. These variables are documented on page 148.)

`\g__file_stack_seq` The input list of files is stored as a sequence stack. In package mode we can recover the information from the details held by L^AT_EX 2_ε (we must be in the preamble and loaded using `\usepackage` or `\RequirePackage`). As L^AT_EX 2_ε doesn't store directory and name separately, we stick to the same convention here.

```

9668 \seq_new:N \g__file_stack_seq
9669 <*package>
9670 \group_begin:
9671 \cs_set_protected:Npn \__file_tmp:w #1#2#3
9672 {
9673   \tl_if_blank:nTF {#1}
9674   {
9675     \cs_set:Npn \__file_tmp:w ##1 " ##2 " ##3 \q_stop
9676     { { } {##2} { } }
9677     \seq_gput_right:Nx \g__file_stack_seq
9678     {
9679       \exp_after:wN \__file_tmp:w \tex_jobname:D
9680       " \tex_jobname:D " \q_stop
9681     }
9682   }

```

```

9683     {
9684         \seq_gput_right:Nn \g__file_stack_seq { { } {#1} {#2} }
9685         \__file_tmp:w
9686     }
9687 }
9688 \cs_if_exist:NT \@currnamestack
9689 { \exp_after:wN \__file_tmp:w \@currnamestack }
9690 \group_end:
9691 \</package>

```

(End definition for `\g__file_stack_seq`.)

`\g__file_record_seq` The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list. The current file name should be included in the file list! In format mode, this is done at the very start of the T_EX run. In package mode we will eventually copy the contents of `\@filelist`.

```

9692 \seq_new:N \g__file_record_seq
9693 \*initex>
9694 \tex_everyjob:D \exp_after:wN
9695 {
9696     \tex_the:D \tex_everyjob:D
9697     \seq_gput_right:NV \g__file_record_seq \g_file_curr_name_str
9698 }
9699 \</initex>

```

(End definition for `\g__file_record_seq`.)

`\l__file_tmp_tl` Used as a short-term scratch variable.

```

9700 \tl_new:N \l__file_tmp_tl

```

(End definition for `\l__file_tmp_tl`.)

`\l__file_base_name_str` For storing the basename and full path whilst passing data internally.

```

\l__file_full_name_str
9701 \str_new:N \l__file_base_name_str
9702 \str_new:N \l__file_full_name_str

```

(End definition for `\l__file_base_name_str` and `\l__file_full_name_str`.)

`\l__file_dir_str` Used in parsing a path into parts: in contrast to the above, these are never used outside of the current module.

```

\l__file_ext_str
\l__file_name_str
9703 \str_new:N \l__file_dir_str
9704 \str_new:N \l__file_ext_str
9705 \str_new:N \l__file_name_str

```

(End definition for `\l__file_dir_str`, `\l__file_ext_str`, and `\l__file_name_str`.)

`\l_file_search_path_seq` The current search path.

```

9706 \seq_new:N \l_file_search_path_seq

```

(End definition for `\l_file_search_path_seq`. This variable is documented on page 149.)

`\l__file_tmp_seq` Scratch space for comma list conversion in package mode.

```

9707 \*package>
9708 \seq_new:N \l__file_tmp_seq
9709 \</package>

```

(End definition for \l__file_tmp_seq.)

_kernel_file_name_sanitiz:nN For converting a token list to a string where active characters are treated as strings from the start. The logic to the quoting normalisation is the same as used by _file_name_quote:nN lualatexquotejobname: check for balanced ", and assuming they balance strip all of _file_name_sanitiz_aux:n them out before quoting the entire name if it contains spaces.

```

9710 \cs_new_protected:Npn \_kernel_file_name_sanitiz:nN #1#2
9711 {
9712   \group_begin:
9713   \seq_map_inline:Nn \l_char_active_seq
9714   {
9715     \tl_set:Nx \l__file_tmp_tl { \iow_char:N ##1 }
9716     \char_set_active_eq:NN ##1 \l__file_tmp_tl
9717   }
9718   \tl_set:Nx \l__file_tmp_tl {#1}
9719   \tl_set:Nx \l__file_tmp_tl
9720   { \tl_to_str:N \l__file_tmp_tl }
9721   \exp_args:NNNV \group_end:
9722   \str_set:Nn #2 \l__file_tmp_tl
9723 }
9724 \cs_new_protected:Npn \_file_name_quote:nN #1#2
9725 {
9726   \str_set:Nx #2 {#1}
9727   \int_if_even:nF
9728   { 0 \tl_map_function:NN #2 \_file_name_quote_aux:n }
9729   {
9730     \_kernel_msg_error:nnx
9731     { kernel } { unbalanced-quote-in-filename } {#2}
9732   }
9733   \tl_remove_all:Nn #2 { " }
9734   \tl_if_in:NnT #2 { ~ }
9735   { \str_set:Nx #2 { " \exp_not:V #2 " } }
9736 }
9737 \cs_new:Npn \_file_name_quote_aux:n #1
9738 { \token_if_eq_charcode:NNT #1 " { + 1 } }

```

(End definition for _kernel_file_name_sanitiz:nN, _file_name_quote:nN, and _file_name_sanitiz_aux:n.)

\file_get_full_name:nN The way to test if a file exists is to try to open it: if it does not exist then T_EX reports end-of-file. A search is made looking at each potential path in turn (starting from the \file_get_full_name:VN current directory). The first location is of course treated as the correct one: this is done by jumping to \prg_break_point:. If nothing is found, #2 is returned empty. A _file_get_full_name_search:nN special case when there is no extension is that once the first location is found we test the existence of the file with .tex extension in that directory, and if it exists we include the .tex extension in the result.

```

9739 \cs_new_protected:Npn \file_get_full_name:nN #1#2
9740 {
9741   \_kernel_file_name_sanitiz:nN {#1} \l__file_base_name_str
9742   \_file_get_full_name_search:nN { } \use:n
9743   \seq_map_inline:Nn \l_file_search_path_seq
9744   { \_file_get_full_name_search:nN { ##1 / } \seq_map_break:n }
9745 }

```

⟨*package⟩

```

9746 \cs_if_exist:NT \input@path
9747 {
9748   \tl_map_inline:Nn \input@path
9749     { \__file_get_full_name_search:nN { ##1 } \tl_map_break:n }
9750 }
9751 \package
9752 \str_clear:N \l__file_full_name_str
9753 \prg_break_point:
9754 \str_if_empty:NF \l__file_full_name_str
9755 {
9756   \exp_args:NV \file_parse_full_name:nNNN \l__file_full_name_str
9757   \l__file_dir_str \l__file_name_str \l__file_ext_str
9758   \str_if_empty:NT \l__file_ext_str
9759   {
9760     \__kernel_ior_open:No \g__file_internal_ior
9761     { \l__file_full_name_str .tex }
9762     \ior_if_eof:NF \g__file_internal_ior
9763     { \str_put_right:Nn \l__file_full_name_str { .tex } }
9764   }
9765 }
9766 \str_set_eq:NN #2 \l__file_full_name_str
9767 \ior_close:N \g__file_internal_ior
9768 }
9769 \cs_generate_variant:Nn \file_get_full_name:nN { V }
9770 \cs_new_protected:Npn \__file_get_full_name_search:nN #1#2
9771 {
9772   \__file_name_quote:nN
9773     { \tl_to_str:n {#1} \l__file_base_name_str }
9774   \l__file_full_name_str
9775   \__kernel_ior_open:No \g__file_internal_ior \l__file_full_name_str
9776   \ior_if_eof:NF \g__file_internal_ior { #2 { \prg_break: } }
9777 }

```

(End definition for `\file_get_full_name:nN` and `__file_get_full_name_search:nN`. This function is documented on page 149.)

`\file_if_exist:nTF` The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path contains something, whereas if the file was not located then the return value is empty.

```

9778 \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
9779 {
9780   \file_get_full_name:nN {#1} \l__file_full_name_str
9781   \str_if_empty:NTF \l__file_full_name_str
9782     { \prg_return_false: }
9783     { \prg_return_true: }
9784 }

```

(End definition for `\file_if_exist:nTF`. This function is documented on page 149.)

`__kernel_file_missing:n` An error message for a missing file, also used in `\ior_open:Nn`.

```

9785 \cs_new_protected:Npn \__kernel_file_missing:n #1
9786 {
9787   \__kernel_file_name_sanitizize:nN {#1} \l__file_base_name_str
9788   \__kernel_msg_error:nnx { kernel } { file-not-found }

```

```

9789     { \l__file_base_name_str }
9790   }

```

(End definition for _kernel_file_missing:n.)

\file_input:n

Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the \g__file_stack_seq, and add it to the file list, either \g__file_record_seq, or \@filelist in package mode.

__file_input:n

__file_input:V

__file_input_push:n

_kernel_file_input_push:n

__file_input_pop:

_kernel_file_input_pop:

__file_input_pop:nnn

```

9791 \cs_new_protected:Npn \file_input:n #1
9792 {
9793   \file_get_full_name:nN {#1} \l__file_full_name_str
9794   \str_if_empty:NTF \l__file_full_name_str
9795     { \_kernel_file_missing:n {#1} }
9796     { \__file_input:V \l__file_full_name_str }
9797 }
9798 \cs_new_protected:Npn \__file_input:n #1
9799 {
9800   < *initex >
9801   \seq_gput_right:Nn \g__file_record_seq {#1}
9802   < /initex >
9803   < *package >
9804   \clist_if_exist:NTF \@filelist
9805     { \@addtofilelist {#1} }
9806     { \seq_gput_right:Nn \g__file_record_seq {#1} }
9807   < /package >
9808   \__file_input_push:n {#1}
9809   \tex_input:D #1 \c_space_tl
9810   \__file_input_pop:
9811 }
9812 \cs_generate_variant:Nn \__file_input:n { V }

```

Keeping a track of the file data is easy enough: we store the separated parts so we do not need to parse them twice.

```

9813 \cs_new_protected:Npn \__file_input_push:n #1
9814 {
9815   \seq_gpush:Nx \g__file_stack_seq
9816   {
9817     { \g_file_curr_dir_str }
9818     { \g_file_curr_name_str }
9819     { \g_file_curr_ext_str }
9820   }
9821   \file_parse_full_name:nNNN {#1}
9822   \l__file_dir_str \l__file_name_str \l__file_ext_str
9823   \str_gset_eq:NN \g_file_curr_dir_str \l__file_dir_str
9824   \str_gset_eq:NN \g_file_curr_name_str \l__file_name_str
9825   \str_gset_eq:NN \g_file_curr_ext_str \l__file_ext_str
9826 }
9827 < *package >
9828 \cs_new_eq:NN \_kernel_file_input_push:n \__file_input_push:n
9829 < /package >
9830 \cs_new_protected:Npn \__file_input_pop:
9831 {
9832   \seq_gpop:NN \g__file_stack_seq \l__file_tmp_tl
9833   \exp_after:wN \__file_input_pop:nnn \l__file_tmp_tl

```

```

9834     }
9835 < *package>
9836 \cs_new_eq:NN \__kernel_file_input_pop: \__file_input_pop:
9837 < /package>
9838 \cs_new_protected:Npn \__file_input_pop:nnn #1#2#3
9839 {
9840     \str_gset:Nn \g_file_curr_dir_str {#1}
9841     \str_gset:Nn \g_file_curr_name_str {#2}
9842     \str_gset:Nn \g_file_curr_ext_str {#3}
9843 }

```

(End definition for `\file_input:n` and others. This function is documented on page 149.)

`\file_parse_full_name:nNNN` Parsing starts by stripping off any surrounding quotes. Then find the directory #4 by splitting at the last /. (The auxiliary returns `true/false` depending on whether it found the delimiter.) We correct for the case of a file in the root /, as in that case we wish to keep the trailing (and only) slash. Then split the base name #5 at the last dot. If there was indeed a dot, #5 contains the name and #6 the extension without the dot, which we add back for convenience. In the special case of no extension given, the auxiliary stored the name into #6, we just have to move it to #5.

```

9844 \cs_new_protected:Npn \file_parse_full_name:nNNN #1#2#3#4
9845 {
9846     \exp_after:wN \__file_parse_full_name_auxi:w
9847     \tl_to_str:n { #1 " #1 " } \q_stop #2#3#4
9848 }
9849 \cs_new_protected:Npn \__file_parse_full_name_auxi:w
9850 #1 " #2 " #3 \q_stop #4#5#6
9851 {
9852     \__file_parse_full_name_split:nNNNTF {#2} / #4 #5
9853     { \str_if_empty:NT #4 { \str_set:Nn #4 { / } } }
9854     { }
9855     \exp_args:No \__file_parse_full_name_split:nNNNTF {#5} . #5 #6
9856     { \str_put_left:Nn #6 { . } }
9857     {
9858         \str_set_eq:NN #5 #6
9859         \str_clear:N #6
9860     }
9861 }
9862 \cs_new_protected:Npn \__file_parse_full_name_split:nNNNTF #1#2#3#4
9863 {
9864     \cs_set_protected:Npn \__file_tmp:w ##1 ##2 #2 ##3 \q_stop
9865     {
9866         \tl_if_empty:nTF {##3}
9867         {
9868             \str_set:Nn #4 {##2}
9869             \tl_if_empty:nTF {##1}
9870             {
9871                 \str_clear:N #3
9872                 \use_ii:nn
9873             }
9874             {
9875                 \str_set:Nx #3 { \str_tail:n {##1} }
9876                 \use_i:nn
9877             }
9878         }
9879     }

```

```

9878     }
9879     { \_file_tmp:w { ##1 ##2 ##3 } ##3 \q_stop }
9880   }
9881   \_file_tmp:w { } #1 #2 \q_stop
9882 }

```

(End definition for `\file_parse_full_name:nNNN`, `_file_parse_full_name_auxi:w`, and `_file_parse_full_name_split:nNNNTF`. This function is documented on page 149.)

`\file_show_list:` A function to list all files used to the log, without duplicates. In package mode, if `\file_log_list:` `\@filelist` is still defined, we need to take this list of file names into account (we capture it `\AtBeginDocument` into `\g__file_record_seq`), turning it to a string (this does not affect the commas of this comma list).

```

9883 \cs_new_protected:Npn \file_show_list: { \_file_list:N \msg_show:nnxxxx }
9884 \cs_new_protected:Npn \file_log_list: { \_file_list:N \msg_log:nnxxxx }
9885 \cs_new_protected:Npn \_file_list:N #1
9886 {
9887   \seq_clear:N \l__file_tmp_seq
9888   \*package
9889   \clist_if_exist:NT \@filelist
9890   {
9891     \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
9892     { \tl_to_str:N \@filelist }
9893   }
9894   \*package
9895   \seq_concat:NNN \l__file_tmp_seq \l__file_tmp_seq \g__file_record_seq
9896   \seq_remove_duplicates:N \l__file_tmp_seq
9897   #1 { LaTeX/kernel } { file-list }
9898   { \seq_map_function:NN \l__file_tmp_seq \_file_list_aux:n }
9899   { } { } { }
9900 }
9901 \cs_new:Npn \_file_list_aux:n #1 { \iow_newline: #1 }

```

(End definition for `\file_show_list:` and others. These functions are documented on page 149.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here. File names recorded in `\@filelist` must be turned to strings before being added to `\g__file_record_seq`.

```

9902 \*package
9903 \AtBeginDocument
9904 {
9905   \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
9906   { \tl_to_str:N \@filelist }
9907   \seq_gconcat:NNN
9908   \g__file_record_seq
9909   \g__file_record_seq
9910   \l__file_tmp_seq
9911 }
9912 \*package

```

18.5 Messages

```

9913 \__kernel_msg_new:nnnn { kernel } { file-not-found }
9914 { File~'##1'~not-found. }
9915 {

```

```

9916     The~requested~file~could~not~be~found~in~the~current~directory,~
9917     in~the~TeX~search~path~or~in~the~LaTeX~search~path.
9918 }
9919 \__kernel_msg_new:nnn { kernel } { file-list }
9920 {
9921     >~File~List~<
9922     #1 \\
9923     .....
9924 }
9925 \__kernel_msg_new:nnnn { kernel } { input-streams-exhausted }
9926 { Input~streams~exhausted }
9927 {
9928     TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
9929     All~16~are~currently~in~use,~and~something~wanted~to~open~
9930     another~one.
9931 }
9932 \__kernel_msg_new:nnnn { kernel } { output-streams-exhausted }
9933 { Output~streams~exhausted }
9934 {
9935     TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
9936     All~16~are~currently~in~use,~and~something~wanted~to~open~
9937     another~one.
9938 }
9939 \__kernel_msg_new:nnnn { kernel } { unbalanced-quote-in-filename }
9940 { Unbalanced~quotes~in~file~name~'#1'. }
9941 {
9942     File~names~must~contain~balanced~numbers~of~quotes~(").
9943 }
9944 \__kernel_msg_new:nnnn { kernel } { iow-indent }
9945 { Only~#1 (arg~1)~allows~#2 }
9946 {
9947     The~command~#2 can~only~be~used~in~messages~
9948     which~will~be~wrapped~using~#1.~
9949     It~was~called~with~argument~'#3'.
9950 }

```

18.6 Deprecated functions

`\g_file_current_name_tl` For removal after 2018-12-31. Contrarily to most other deprecated commands this is expandable so we need to put code by hand in two token lists. We use `\tex_def:D` directly because `\g_file_current_name_tl` is made outer by `\debug_deprecation_on:.`

```

9951 \tl_new:N \g_file_current_name_tl
9952 \tl_gset:Nn \g_file_current_name_tl { \g_file_curr_name_str }
9953 \__kernel_deprecation_code:nn
9954 {
9955     \__kernel_deprecation_error:Nnn \g_file_current_name_tl
9956     { \g_file_curr_name_str } { 2018-12-31 }
9957 }
9958 { \tex_def:D \g_file_current_name_tl { \g_file_curr_name_str } }

```

(End definition for `\g_file_current_name_tl`.)

`\file_path_include:n` Wrapper functions to manage the search path.
`\file_path_remove:n`


```

9959 \__kernel_patch_deprecation:nnNNpn { 2018-12-31 }
9960 { \seq_put_right:Nn \l_file_search_path_seq }
9961 \cs_new_protected:Npn \file_path_include:n #1
9962 {
9963   \__kernel_file_name_sanitiz:n {#1} \l_file_full_name_str
9964   \seq_if_in:NVF \l_file_search_path_seq \l_file_full_name_str
9965   { \seq_put_right:NV \l_file_search_path_seq \l_file_full_name_str }
9966 }
9967 \__kernel_patch_deprecation:nnNNpn { 2018-12-31 }
9968 { \seq_remove_all:Nn \l_file_search_path_seq }
9969 \cs_new_protected:Npn \file_path_remove:n #1
9970 {
9971   \__kernel_file_name_sanitiz:n {#1} \l_file_full_name_str
9972   \seq_remove_all:NV \l_file_search_path_seq \l_file_full_name_str
9973 }

```

(End definition for \file_path_include:n and \file_path_remove:n.)

\file_add_path:nN For removal after 2018-12-31.

```

9974 \__kernel_patch_deprecation:nnNNpn { 2018-12-31 } { \file_get_full_name:nN }
9975 \cs_new_protected:Npn \file_add_path:nN #1#2
9976 {
9977   \file_get_full_name:nN {#1} #2
9978   \str_if_empty:NT #2
9979   { \tl_set:Nn #2 { \q_no_value } }
9980 }

```

(End definition for \file_add_path:nN.)

\file_list: Renamed to \file_log_list:. For removal after 2018-12-31.

```

9981 \__kernel_patch_deprecation:nnNNpn { 2018-12-31 } { \file_log_list: }
9982 \cs_new_protected:Npn \file_list: { \file_log_list: }

```

(End definition for \file_list:.)

\ior_list_streams: These got a more consistent naming.

```

\ior_log_streams: 9983 \__kernel_patch_deprecation:nnNNpn { 2018-12-31 } { \ior_show_list: }
\ior_list_streams: 9984 \cs_new_protected:Npn \ior_list_streams: { \ior_show_list: }
\ior_log_streams: 9985 \__kernel_patch_deprecation:nnNNpn { 2018-12-31 } { \ior_log_list: }
\ior_log_streams: 9986 \cs_new_protected:Npn \ior_log_streams: { \ior_log_list: }
9987 \__kernel_patch_deprecation:nnNNpn { 2018-12-31 } { \iow_show_list: }
9988 \cs_new_protected:Npn \iow_list_streams: { \iow_show_list: }
9989 \__kernel_patch_deprecation:nnNNpn { 2018-12-31 } { \iow_log_list: }
9990 \cs_new_protected:Npn \iow_log_streams: { \iow_log_list: }

```

(End definition for \ior_list_streams: and others.)

```

9991 </initex | package>

```

19 l3skip implementation

```
9992 \*initex | package)
```

```
9993 \@@=dim)
```

19.1 Length primitives renamed

\if_dim:w Primitives renamed.

```
\__dim_eval:w 9994 \cs_new_eq:NN \if_dim:w \tex_ifdim:D
\__dim_eval_end: 9995 \cs_new_eq:NN \__dim_eval:w \etex_dimexpr:D
9996 \cs_new_eq:NN \__dim_eval_end: \tex_relax:D
```

(End definition for `\if_dim:w`, `__dim_eval:w`, and `__dim_eval_end:`. This function is documented on page 164.)

19.2 Creating and initialising dim variables

\dim_new:N Allocating $\langle dim \rangle$ registers ...

```
\dim_new:c 9997 \*package)
9998 \cs_new_protected:Npn \dim_new:N #1
9999 {
10000 \__kernel_chk_if_free_cs:N #1
10001 \cs:w newdimen \cs_end: #1
10002 }
10003 \</package>
10004 \cs_generate_variant:Nn \dim_new:N { c }
```

(End definition for `\dim_new:N`. This function is documented on page 150.)

\dim_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants. We cannot use `\dim_gset:Nn` because debugging code would complain that the constant is not a global variable. Since `\dim_const:Nn` does not need to be fast, use `\dim_eval:n` to avoid needing a debugging patch that wraps the expression in checking code.

```
10005 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN c #1 } { }
10006 \cs_new_protected:Npn \dim_const:Nn #1#2
10007 {
10008 \dim_new:N #1
10009 \tex_global:D #1 ~ \dim_eval:n {#2} \scan_stop:
10010 }
10011 \cs_generate_variant:Nn \dim_const:Nn { c }
```

(End definition for `\dim_const:Nn`. This function is documented on page 150.)

\dim_zero:N Reset the register to zero. Using `\c_zero_skip` deals with the case where the variable passed is incorrectly a skip (for example a L^AT_EX 2_ε length).

\dim_zero:c

\dim_gzero:N

\dim_gzero:c

```
10012 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
10013 \cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_skip }
10014 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
10015 \cs_new_protected:Npn \dim_gzero:N #1
10016 { \tex_global:D #1 \c_zero_skip }
10017 \cs_generate_variant:Nn \dim_zero:N { c }
10018 \cs_generate_variant:Nn \dim_gzero:N { c }
```

(End definition for `\dim_zero:N` and `\dim_gzero:N`. These functions are documented on page 150.)

```

\dim_zero_new:N Create a register if needed, otherwise clear it.
\dim_zero_new:c 10019 \cs_new_protected:Npn \dim_zero_new:N #1
\dim_gzero_new:N 10020 { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\dim_gzero_new:c 10021 \cs_new_protected:Npn \dim_gzero_new:N #1
10022 { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
10023 \cs_generate_variant:Nn \dim_zero_new:N { c }
10024 \cs_generate_variant:Nn \dim_gzero_new:N { c }

(End definition for \dim_zero_new:N and \dim_gzero_new:N. These functions are documented on page
150.)

```

```

\dim_if_exist_p:N Copies of the cs functions defined in l3basics.
\dim_if_exist_p:c 10025 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
\dim_if_exist:NTF 10026 { TF , T , F , p }
\dim_if_exist:cTF 10027 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
10028 { TF , T , F , p }

(End definition for \dim_if_exist:NTF. This function is documented on page 150.)

```

19.3 Setting dim variables

Several functions here have a signature `:Nn` and are such that when debugging, the first argument should be checked to be a local/global variable and the second should be wrapped in code for an expression. The temporary function `__dim_tmp:w` finds the name `#3` of the function being redefined and writes the appropriate patch.

```

10029 \cs_set_protected:Npn \__dim_tmp:w #1#2#3
10030 {
10031   \__kernel_patch_args:nnnNnNpn
10032     { #1 ##1 }
10033     { }
10034     { {##1} { \__kernel_chk_expr:nNnN {##2} \__dim_eval:w { } #3 } }
10035     #2 #3
10036 }

```

```

\dim_set:Nn Setting dimensions is easy enough but when debugging we want both to check that the
\dim_set:cn variable is correctly local/global and to wrap the expression in some code. The \scan_
\dim_gset:Nn stop: deals with the case where the variable passed is a skip (for example a LATEX 2ε
\dim_gset:cn length).

```

```

10037 \__dim_tmp:w \__kernel_chk_var_local:N
10038 \cs_new_protected:Npn \dim_set:Nn #1#2
10039 { #1 ~ \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
10040 \__dim_tmp:w \__kernel_chk_var_global:N
10041 \cs_new_protected:Npn \dim_gset:Nn #1#2
10042 { \tex_global:D #1 ~ \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
10043 \cs_generate_variant:Nn \dim_set:Nn { c }
10044 \cs_generate_variant:Nn \dim_gset:Nn { c }

```

(End definition for `\dim_set:Nn` and `\dim_gset:Nn`. These functions are documented on page 151.)

```

\dim_set_eq:NN All straightforward, with a \scan_stop: to deal with the case where #1 is (incorrectly)
\dim_set_eq:cN a skip.
\dim_set_eq:Nc 10045 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\dim_set_eq:cc 10046 \cs_new_protected:Npn \dim_set_eq:NN #1#2
\dim_gset_eq:NN
\dim_gset_eq:cN
\dim_gset_eq:Nc
\dim_gset_eq:cc

```

```

10047 { #1 = #2 \scan_stop: }
10048 \cs_generate_variant:Nn \dim_set_eq:NN { c , Nc , cc }
10049 \__kernel_patch:nNnpn { \__kernel_chk_var_global:N #1 } { }
10050 \cs_new_protected:Npn \dim_gset_eq:NN #1#2
10051 { \tex_global:D #1 = #2 \scan_stop: }
10052 \cs_generate_variant:Nn \dim_gset_eq:NN { c , Nc , cc }

```

(End definition for `\dim_set_eq:NN` and `\dim_gset_eq:NN`. These functions are documented on page 151.)

`\dim_add:Nn` Using by here deals with the (incorrect) case `\dimen123`. Using `\scan_stop:` deals with
`\dim_add:cn` skip variables. Since debugging checks that the variable is correctly local/global, the
`\dim_gadd:Nn` global versions cannot be defined as `\tex_global:D` followed by the local versions. The
`\dim_gadd:cn` debugging code is inserted by `__dim_tmp:w`.

```

\dim_sub:Nn 10053 \__dim_tmp:w \__kernel_chk_var_local:N
\dim_sub:cn 10054 \cs_new_protected:Npn \dim_add:Nn #1#2
\dim_gsub:Nn 10055 { \tex_advance:D #1 by \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
\dim_gsub:cn 10056 \__dim_tmp:w \__kernel_chk_var_global:N
10057 \cs_new_protected:Npn \dim_gadd:Nn #1#2
10058 {
10059 \tex_global:D \tex_advance:D #1 by
10060 \__dim_eval:w #2 \__dim_eval_end: \scan_stop:
10061 }
10062 \cs_generate_variant:Nn \dim_add:Nn { c }
10063 \cs_generate_variant:Nn \dim_gadd:Nn { c }
10064 \__dim_tmp:w \__kernel_chk_var_local:N
10065 \cs_new_protected:Npn \dim_sub:Nn #1#2
10066 { \tex_advance:D #1 by - \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
10067 \__dim_tmp:w \__kernel_chk_var_global:N
10068 \cs_new_protected:Npn \dim_gsub:Nn #1#2
10069 {
10070 \tex_global:D \tex_advance:D #1 by
10071 - \__dim_eval:w #2 \__dim_eval_end: \scan_stop:
10072 }
10073 \cs_generate_variant:Nn \dim_sub:Nn { c }
10074 \cs_generate_variant:Nn \dim_gsub:Nn { c }

```

(End definition for `\dim_add:Nn` and others. These functions are documented on page 151.)

19.4 Utilities for dimension calculations

`\dim_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value
`__dim_abs:N` is evaluated by removing a leading - if present.

```

\dim_max:nn 10075 \__kernel_patch_args:nNnpn
\dim_min:nn 10076 { { \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_abs:n } }
\__dim_maxmin:wwN 10077 \cs_new:Npn \dim_abs:n #1
10078 {
10079 \exp_after:wN \__dim_abs:N
10080 \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
10081 }
10082 \cs_new:Npn \__dim_abs:N #1
10083 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
10084 \__kernel_patch_args:nNnpn
10085 {

```

```

10086 { \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_max:nn }
10087 { \__kernel_chk_expr:nNnN {#2} \__dim_eval:w { } \dim_max:nn }
10088 }
10089 \cs_new:Npn \dim_max:nn #1#2
10090 {
10091   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
10092   \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
10093   \dim_use:N \__dim_eval:w #2 ;
10094   >
10095   \__dim_eval_end:
10096 }
10097 \__kernel_patch_args:nNnNpn
10098 {
10099   { \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_min:nn }
10100   { \__kernel_chk_expr:nNnN {#2} \__dim_eval:w { } \dim_min:nn }
10101 }
10102 \cs_new:Npn \dim_min:nn #1#2
10103 {
10104   \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
10105   \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
10106   \dim_use:N \__dim_eval:w #2 ;
10107   <
10108   \__dim_eval_end:
10109 }
10110 \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
10111 {
10112   \if_dim:w #1 #3 #2 ~
10113     #1
10114   \else:
10115     #2
10116   \fi:
10117 }

```

(End definition for `\dim_abs:n` and others. These functions are documented on page 151.)

`\dim_ratio:nn` With dimension expressions, something like `10 pt * (5 pt / 10 pt)` does not work. `__dim_ratio:n` Instead, the ratio part needs to be converted to an integer expression. Using `\int_value:w` forces everything into `sp`, avoiding any decimal parts.

```

10118 \cs_new:Npn \dim_ratio:nn #1#2
10119 { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
10120 \cs_new:Npn \__dim_ratio:n #1
10121 { \int_value:w \__dim_eval:w (#1) \__dim_eval_end: }

```

(End definition for `\dim_ratio:nn` and `__dim_ratio:n`. This function is documented on page 152.)

19.5 Dimension expression conditionals

`\dim_compare_p:nNn` Simple comparison.

`\dim_compare:nNnTF`

```

10122 \__kernel_patch_conditional_args:nNnNpn
10123 {
10124   { \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_compare:nNn }
10125   { \__dim_eval_end: #2 }
10126   { \__kernel_chk_expr:nNnN {#3} \__dim_eval:w { } \dim_compare:nNn }
10127 }

```

```

10128 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
10129 {
10130   \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
10131     \prg_return_true: \else: \prg_return_false: \fi:
10132 }

```

(End definition for `\dim_compare:nNnTF`. This function is documented on page 152.)

`\dim_compare_p:n` This code is adapted from the `\int_compare:nTF` function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing `__dim_compare_error:`. Just like for integers, the looping auxiliary `__dim_compare:wNn` closes a primitive conditional and opens a new one. It is actually easier to grab a dimension operand than an integer one, because once evaluated, dimensions all end with `pt` (with category other). Thus we do not need specific auxiliaries for the three “simple” relations `<`, `=`, and `>`.

```

\__dim_compare:w
\__dim_compare:wNn
\__dim_compare_=w
\__dim_compare_!=w
\__dim_compare_<w
\__dim_compare_>w
\__dim_compare_error:
10133 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
10134 {
10135   \exp_after:wN \__dim_compare:w
10136   \dim_use:N \__dim_eval:w #1 \__dim_compare_error:
10137 }
10138 \cs_new:Npn \__dim_compare:w #1 \__dim_compare_error:
10139 {
10140   \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
10141   \__dim_compare:wNn #1 ? { = \__dim_compare_end:w \else: } \q_stop
10142 }
10143 \exp_args:Nno \use:nn
10144 { \cs_new:Npn \__dim_compare:wNn #1 } { \tl_to_str:n {pt} #2#3 }
10145 {
10146   \if_meaning:w = #3
10147     \use:c { __dim_compare_#2:w }
10148   \fi:
10149   #1 pt \exp_stop_f:
10150   \prg_return_false:
10151   \exp_after:wN \use_none_delimit_by_q_stop:w
10152   \fi:
10153   \reverse_if:N \if_dim:w #1 pt #2
10154   \exp_after:wN \__dim_compare:wNn
10155   \dim_use:N \__dim_eval:w #3
10156 }
10157 \cs_new:cpn { __dim_compare_ ! :w }
10158   #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
10159 \cs_new:cpn { __dim_compare_ = :w }
10160   #1 \__dim_eval:w = { #1 \__dim_eval:w }
10161 \cs_new:cpn { __dim_compare_ < :w }
10162   #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
10163 \cs_new:cpn { __dim_compare_ > :w }
10164   #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
10165 \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \q_stop
10166 { #1 \prg_return_false: \else: \prg_return_true: \fi: }
10167 \cs_new_protected:Npn \__dim_compare_error:
10168 {
10169   \if_int_compare:w \c_zero \c_zero \fi:
10170   =
10171   \__dim_compare_error:

```

```
10172 }
```

(End definition for `\dim_compare:nTF` and others. This function is documented on page 153.)

```

\dim_case:nn For dimension cases, the first task to fully expand the check condition. The over all idea
\dim_case:nnTF is then much the same as for \str_case:nn(TF) as described in l3basics.
\__dim_case:nnTF 10173 \cs_new:Npn \dim_case:nnTF #1
\__dim_case:nw 10174 {
\__dim_case_end:nw 10175 \exp:w
10176 \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
10177 }
10178 \cs_new:Npn \dim_case:nnT #1#2#3
10179 {
10180 \exp:w
10181 \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
10182 }
10183 \cs_new:Npn \dim_case:nnF #1#2
10184 {
10185 \exp:w
10186 \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
10187 }
10188 \cs_new:Npn \dim_case:nn #1#2
10189 {
10190 \exp:w
10191 \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
10192 }
10193 \cs_new:Npn \__dim_case:nnTF #1#2#3#4
10194 { \__dim_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
10195 \cs_new:Npn \__dim_case:nw #1#2#3
10196 {
10197 \dim_compare:nNnTF {#1} = {#2}
10198 { \__dim_case_end:nw {#3} }
10199 { \__dim_case:nw {#1} }
10200 }
10201 \cs_new:Npn \__dim_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
10202 { \exp_end: #1 #4 }
```

(End definition for `\dim_case:nnTF` and others. This function is documented on page 154.)

19.6 Dimension expression loops

`\dim_while_do:nn` while_do and do_while functions for dimensions. Same as for the int type only the names have changed.

```

\dim_while_do:nn 10203 \cs_new:Npn \dim_while_do:nn #1#2
\dim_until_do:nn 10204 {
\dim_do_while:nn 10205 \dim_compare:nT {#1}
10206 {
10207 #2
10208 \dim_while_do:nn {#1} {#2}
10209 }
10210 }
10211 \cs_new:Npn \dim_until_do:nn #1#2
10212 {
10213 \dim_compare:nF {#1}
```

```

10214     {
10215         #2
10216         \dim_until_do:nn {#1} {#2}
10217     }
10218 }
10219 \cs_new:Npn \dim_do_while:nn #1#2
10220 {
10221     #2
10222     \dim_compare:nT {#1}
10223     { \dim_do_while:nn {#1} {#2} }
10224 }
10225 \cs_new:Npn \dim_do_until:nn #1#2
10226 {
10227     #2
10228     \dim_compare:nF {#1}
10229     { \dim_do_until:nn {#1} {#2} }
10230 }

```

(End definition for `\dim_while_do:nn` and others. These functions are documented on page 155.)

`\dim_while_do:nNnn` `while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```

\dim_until_do:nNnn
\dim_do_while:nNnn
\dim_do_until:nNnn
10231 \cs_new:Npn \dim_while_do:nNnn #1#2#3#4
10232 {
10233     \dim_compare:nNnT {#1} #2 {#3}
10234     {
10235         #4
10236         \dim_while_do:nNnn {#1} #2 {#3} {#4}
10237     }
10238 }
10239 \cs_new:Npn \dim_until_do:nNnn #1#2#3#4
10240 {
10241     \dim_compare:nNnF {#1} #2 {#3}
10242     {
10243         #4
10244         \dim_until_do:nNnn {#1} #2 {#3} {#4}
10245     }
10246 }
10247 \cs_new:Npn \dim_do_while:nNnn #1#2#3#4
10248 {
10249     #4
10250     \dim_compare:nNnT {#1} #2 {#3}
10251     { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
10252 }
10253 \cs_new:Npn \dim_do_until:nNnn #1#2#3#4
10254 {
10255     #4
10256     \dim_compare:nNnF {#1} #2 {#3}
10257     { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
10258 }

```

(End definition for `\dim_while_do:nNnn` and others. These functions are documented on page 155.)

19.7 Dimension step functions

`\dim_step_function:nnnN`

`__dim_step:wwwN`

`__dim_step:NnnnN`

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

10259 \__kernel_patch_args:nNnpn
10260 {
10261   {
10262     \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { }
10263     \dim_step_function:nnnN
10264   }
10265   {
10266     \__kernel_chk_expr:nNnN {#2} \__dim_eval:w { }
10267     \dim_step_function:nnnN
10268   }
10269   {
10270     \__kernel_chk_expr:nNnN {#3} \__dim_eval:w { }
10271     \dim_step_function:nnnN
10272   }
10273 }
10274 \cs_new:Npn \dim_step_function:nnnN #1#2#3
10275 {
10276   \exp_after:wN \__dim_step:wwwN
10277   \tex_the:D \__dim_eval:w #1 \exp_after:wN ;
10278   \tex_the:D \__dim_eval:w #2 \exp_after:wN ;
10279   \tex_the:D \__dim_eval:w #3 ;
10280 }
10281 \cs_new:Npn \__dim_step:wwwN #1; #2; #3; #4
10282 {
10283   \dim_compare:nNnTF {#2} > \c_zero_dim
10284   { \__dim_step:NnnnN > }
10285   {
10286     \dim_compare:nNnTF {#2} = \c_zero_dim
10287     {
10288       \__kernel_msg_expandable_error:nnn { kernel } { zero-step } {#4}
10289       \use_none:nnnn
10290     }
10291     { \__dim_step:NnnnN < }
10292   }
10293   {#1} {#2} {#3} #4
10294 }
10295 \cs_new:Npn \__dim_step:NnnnN #1#2#3#4#5
10296 {
10297   \dim_compare:nNnF {#2} #1 {#4}
10298   {
10299     #5 {#2}
10300     \exp_args:NNf \__dim_step:NnnnN
10301     #1 { \dim_eval:n { #2 + #3 } } {#3} {#4} #5
10302   }
10303 }
```

(End definition for `\dim_step_function:nnnN`, `__dim_step:wwwN`, and `__dim_step:NnnnN`. This func-

tion is documented on page 155.)

`\dim_step_inline:nnnn` The approach here is to build a function, with a global integer required to make the
`\dim_step_variable:nnnNn` nesting safe (as seen in other in line functions), and map that function using `\dim-`
`__dim_step:NNnnnn` `step_function:nnnN`. We put a `\prg_break_point:Nn` so that map_break functions
from other modules correctly decrement `\g__kernel_prg_map_int` before looking for
their own break point. The first argument is `\scan_stop:`, so that no breaking function
recognizes this break point as its own.

```

10304 \cs_new_protected:Npn \dim_step_inline:nnnn
10305 {
10306   \int_gincr:N \g__kernel_prg_map_int
10307   \exp_args:NNc \__dim_step:NNnnnn
10308   \cs_gset_protected:Npn
10309     { \__dim_map_ \int_use:N \g__kernel_prg_map_int :w }
10310 }
10311 \cs_new_protected:Npn \dim_step_variable:nnnNn #1#2#3#4#5
10312 {
10313   \int_gincr:N \g__kernel_prg_map_int
10314   \exp_args:NNc \__dim_step:NNnnnn
10315   \cs_gset_protected:Npx
10316     { \__dim_map_ \int_use:N \g__kernel_prg_map_int :w }
10317     {#1}{#2}{#3}
10318     {
10319       \tl_set:Nn \exp_not:N #4 {##1}
10320       \exp_not:n {#5}
10321     }
10322 }
10323 \cs_new_protected:Npn \__dim_step:NNnnnn #1#2#3#4#5#6
10324 {
10325   #1 #2 ##1 {#6}
10326   \dim_step_function:nnnN {#3} {#4} {#5} #2
10327   \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
10328 }

```

(End definition for `\dim_step_inline:nnnn`, `\dim_step_variable:nnnNn`, and `__dim_step:NNnnnn`.
These functions are documented on page 155.)

19.8 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

10329 \__kernel_patch_args:nNNpn
10330 { { \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_eval:n } }
10331 \cs_new:Npn \dim_eval:n #1
10332 { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 156.)

`\dim_use:N` Accessing a $\langle dim \rangle$.

`\dim_use:c` 10333 \cs_new_eq:NN \dim_use:N \tex_the:D

We hand-code this for some speed gain:

```

10334 %\cs_generate_variant:Nn \dim_use:N { c }
10335 \cs_new:Npn \dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End definition for `\dim_use:N`. This function is documented on page 156.)

`\dim_to_decimal:n` A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression `#1` then remove the trailing `pt`. When debugging is enabled, the argument is put in parentheses as this prevents the dimension expression from terminating early and leaving extra tokens lying around. This is used a lot by low-level manipulations.

`__dim_to_decimal:w`

```

10336 \__kernel_patch_args:nNNpn
10337 { { \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_to_decimal:n } }
10338 \cs_new:Npn \dim_to_decimal:n #1
10339 {
10340   \exp_after:wN
10341   \__dim_to_decimal:w \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
10342 }
10343 \use:x
10344 {
10345   \cs_new:Npn \exp_not:N \__dim_to_decimal:w
10346     ##1 . ##2 \tl_to_str:n { pt }
10347 }
10348 {
10349   \int_compare:nNnTF {#2} > { 0 }
10350   { #1 . #2 }
10351   { #1 }
10352 }

```

(End definition for `\dim_to_decimal:n` and `__dim_to_decimal:w`. This function is documented on page 156.)

`\dim_to_decimal_in_bp:n` Conversion to big points is done using a scaling inside `__dim_eval:w` as ε -TeX does that using 64-bit precision. Here, 800/803 is the integer fraction for 72/72.27. This is a common case so is hand-coded for accuracy (and speed).

```

10353 \cs_new:Npn \dim_to_decimal_in_bp:n #1
10354 { \dim_to_decimal:n { ( #1 ) * 800 / 803 } }

```

(End definition for `\dim_to_decimal_in_bp:n`. This function is documented on page 157.)

`\dim_to_decimal_in_sp:n` Another hard-coded conversion: this one is necessary to avoid things going off-scale.

```

10355 \__kernel_patch_args:nNNpn
10356 {
10357   {
10358     \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { }
10359     \dim_to_decimal_in_sp:n
10360   }
10361 }
10362 \cs_new:Npn \dim_to_decimal_in_sp:n #1
10363 { \int_value:w \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_to_decimal_in_sp:n`. This function is documented on page 157.)

`\dim_to_decimal_in_unit:nn` An analogue of `\dim_ratio:nn` that produces a decimal number as its result, rather than a rational fraction for use within dimension expressions.

```

10364 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
10365 {
10366   \dim_to_decimal:n
10367   {
10368     1pt *
10369     \dim_ratio:nn {#1} {#2}

```

```

10370     }
10371 }

```

(End definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page 157.)

`\dim_to_fp:n` Defined in `l3fp-convert`, documented here.

(End definition for `\dim_to_fp:n`. This function is documented on page 157.)

19.9 Viewing dim variables

`\dim_show:N` Diagnostics.

```

\dim_show:c 10372 \cs_new_eq:NN \dim_show:N \__kernel_register_show:N
10373 \cs_generate_variant:Nn \dim_show:N { c }

```

(End definition for `\dim_show:N`. This function is documented on page 157.)

`\dim_show:n` Diagnostics. We don't use the \TeX primitive `\showthe` to show dimension expressions: this gives a more unified output.

```

10374 \cs_new_protected:Npn \dim_show:n
10375 { \msg_show_eval:Nn \dim_eval:n }

```

(End definition for `\dim_show:n`. This function is documented on page 158.)

`\dim_log:N` Diagnostics. Redirect output of `\dim_show:n` to the log.

```

\dim_log:c 10376 \cs_new_eq:NN \dim_log:N \__kernel_register_log:N
\dim_log:n 10377 \cs_new_eq:NN \dim_log:c \__kernel_register_log:c
10378 \cs_new_protected:Npn \dim_log:n
10379 { \msg_log_eval:Nn \dim_eval:n }

```

(End definition for `\dim_log:N` and `\dim_log:n`. These functions are documented on page 158.)

19.10 Constant dimensions

`\c_zero_dim` Constant dimensions.

```

\c_max_dim 10380 \dim_const:Nn \c_zero_dim { 0 pt }
10381 \dim_const:Nn \c_max_dim { 16383.99999 pt }

```

(End definition for `\c_zero_dim` and `\c_max_dim`. These variables are documented on page 158.)

19.11 Scratch dimensions

`\l_tmpa_dim` We provide two local and two global scratch registers, maybe we need more or less.

```

\l_tmpb_dim 10382 \dim_new:N \l_tmpa_dim
\g_tmpa_dim 10383 \dim_new:N \l_tmpb_dim
\g_tmpb_dim 10384 \dim_new:N \g_tmpa_dim
10385 \dim_new:N \g_tmpb_dim

```

(End definition for `\l_tmpa_dim` and others. These variables are documented on page 158.)

19.12 Creating and initialising skip variables

10386 <@@=skip>

\skip_new:N Allocation of a new internal registers.

\skip_new:c

```

10387 <*package>
10388 \cs_new_protected:Npn \skip_new:N #1
10389 {
10390     \__kernel_chk_if_free_cs:N #1
10391     \cs:w newskip \cs_end: #1
10392 }
10393 </package>
10394 \cs_generate_variant:Nn \skip_new:N { c }
```

(End definition for `\skip_new:N`. This function is documented on page 158.)

\skip_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants. See **\dim_const:Nn** for why we cannot use **\skip_gset:Nn**.

\skip_const:cn

```

10395 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN c #1 } { }
10396 \cs_new_protected:Npn \skip_const:Nn #1#2
10397 {
10398     \skip_new:N #1
10399     \tex_global:D #1 ~ \skip_eval:n {#2} \scan_stop:
10400 }
10401 \cs_generate_variant:Nn \skip_const:Nn { c }
```

(End definition for `\skip_const:Nn`. This function is documented on page 158.)

\skip_zero:N Reset the register to zero.

\skip_zero:c

```

10402 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\skip_gzero:N
10403 \cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
\skip_gzero:c
10404 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
10405 \cs_new_protected:Npn \skip_gzero:N #1 { \tex_global:D #1 \c_zero_skip }
10406 \cs_generate_variant:Nn \skip_zero:N { c }
10407 \cs_generate_variant:Nn \skip_gzero:N { c }
```

(End definition for `\skip_zero:N` and `\skip_gzero:N`. These functions are documented on page 159.)

\skip_zero_new:N Create a register if needed, otherwise clear it.

\skip_zero_new:c

```

10408 \cs_new_protected:Npn \skip_zero_new:N #1
\skip_gzero_new:N
10409 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\skip_gzero_new:c
10410 \cs_new_protected:Npn \skip_gzero_new:N #1
10411 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
10412 \cs_generate_variant:Nn \skip_zero_new:N { c }
10413 \cs_generate_variant:Nn \skip_gzero_new:N { c }
```

(End definition for `\skip_zero_new:N` and `\skip_gzero_new:N`. These functions are documented on page 159.)

\skip_if_exist_p:N Copies of the cs functions defined in l3basics.

\skip_if_exist_p:c

```

10414 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
\skip_if_exist:NTF
10415 { TF , T , F , p }
\skip_if_exist:cTF
10416 \prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
10417 { TF , T , F , p }
```

(End definition for `\skip_if_exist:NTF`. This function is documented on page 159.)

19.13 Setting skip variables

Much as for dim variables, `__skip_tmp:w` prepares a patch for `:Nn` function definitions in which the first argument should be checked to be a local/global variable and the second should be wrapped in code for an expression.

```

10418 \cs_set_protected:Npn \__skip_tmp:w #1#2#3
10419 {
10420   \__kernel_patch_args:nnnNnNpn
10421   { #1 ##1 }
10422   { }
10423   { {##1} { \__kernel_chk_expr:nNnN {##2} \etex_glueexpr:D { } #3 } }
10424   #2 #3
10425 }
```

`\skip_set:Nn` Much the same as for dimensions.

```

\skip_set:cn 10426 \__skip_tmp:w \__kernel_chk_var_local:N
\skip_gset:Nn 10427 \cs_new_protected:Npn \skip_set:Nn #1#2
\skip_gset:cn 10428 { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
10429 \__skip_tmp:w \__kernel_chk_var_global:N
10430 \cs_new_protected:Npn \skip_gset:Nn #1#2
10431 { \tex_global:D #1 ~ \etex_glueexpr:D #2 \scan_stop: }
10432 \cs_generate_variant:Nn \skip_set:Nn { c }
10433 \cs_generate_variant:Nn \skip_gset:Nn { c }
```

(End definition for `\skip_set:Nn` and `\skip_gset:Nn`. These functions are documented on page 159.)

`\skip_set_eq:NN` All straightforward.

```

\skip_set_eq:cn 10434 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\skip_set_eq:Nc 10435 \cs_generate_variant:Nn \skip_set_eq:NN { c , Nc , cc }
\skip_set_eq:cc 10436 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\skip_gset_eq:Nn 10437 \cs_generate_variant:Nn \skip_gset_eq:NN { c , Nc , cc }
```

`\skip_gset_eq:cn`
`\skip_gset_eq:Nc`
`\skip_gset_eq:cc` (End definition for `\skip_set_eq:NN` and `\skip_gset_eq:NN`. These functions are documented on page 159.)

`\skip_add:Nn` Using by here deals with the (incorrect) case `\skip123`.

```

\skip_add:cn 10438 \__skip_tmp:w \__kernel_chk_var_local:N
\skip_gadd:Nn 10439 \cs_new_protected:Npn \skip_add:Nn #1#2
\skip_gadd:cn 10440 { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
\skip_sub:Nn 10441 \__skip_tmp:w \__kernel_chk_var_global:N
\skip_sub:cn 10442 \cs_new_protected:Npn \skip_gadd:Nn #1#2
\skip_gsub:Nn 10443 { \tex_global:D \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
\skip_gsub:cn 10444 \cs_generate_variant:Nn \skip_add:Nn { c }
10445 \cs_generate_variant:Nn \skip_gadd:Nn { c }
10446 \__skip_tmp:w \__kernel_chk_var_local:N
10447 \cs_new_protected:Npn \skip_sub:Nn #1#2
10448 { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
10449 \__skip_tmp:w \__kernel_chk_var_global:N
10450 \cs_new_protected:Npn \skip_gsub:Nn #1#2
10451 { \tex_global:D \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
10452 \cs_generate_variant:Nn \skip_sub:Nn { c }
10453 \cs_generate_variant:Nn \skip_gsub:Nn { c }
```

(End definition for `\skip_add:Nn` and others. These functions are documented on page 159.)

19.14 Skip expression conditionals

`\skip_if_eq_p:nn` Comparing skips means doing two expansions to make strings, and then testing them.
`\skip_if_eq:nnTF` As a result, only equality is tested.

```

10454 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
10455 {
10456   \str_if_eq_x:nnTF { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
10457   { \prg_return_true: }
10458   { \prg_return_false: }
10459 }
```

(End definition for `\skip_if_eq:nnTF`. This function is documented on page 160.)

`\skip_if_finite_p:n` With ε -TEX, we have an easy access to the order of infinities of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.

```

10460 \cs_set_protected:Npn \__skip_tmp:w #1
10461 {
10462   \__kernel_patch_conditional_args:nNNpnn
10463   {
10464     {
10465       \__kernel_chk_expr:nNnN
10466       {##1} \etex_glueexpr:D { } \skip_if_finite:n
10467     }
10468   }
10469   \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
10470   {
10471     \exp_after:wN \__skip_if_finite:wwNw
10472     \skip_use:N \etex_glueexpr:D ##1 ; \prg_return_false:
10473     #1 ; \prg_return_true: \q_stop
10474   }
10475   \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
10476 }
10477 \exp_args:No \__skip_tmp:w { \tl_to_str:n { fil } }
```

(End definition for `\skip_if_finite:nTF` and `__skip_if_finite:wwNw`. This function is documented on page 160.)

19.15 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```

10478 \__kernel_patch_args:nNNpnn
10479 { { \__kernel_chk_expr:nNnN {#1} \etex_glueexpr:D { } \skip_eval:n } }
10480 \cs_new:Npn \skip_eval:n #1
10481 { \skip_use:N \etex_glueexpr:D #1 \scan_stop: }
```

(End definition for `\skip_eval:n`. This function is documented on page 160.)

`\skip_use:N` Accessing a $\langle skip \rangle$.

```

10482 \cs_new_eq:NN \skip_use:N \tex_the:D
10483 %\cs_generate_variant:Nn \skip_use:N { c }
10484 \cs_new:Npn \skip_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(End definition for `\skip_use:N`. This function is documented on page 160.)

19.16 Inserting skips into the output

```

\skip_horizontal:N Inserting skips.
\skip_horizontal:c
\skip_horizontal:n 10485 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
\skip_vertical:N 10486 \__kernel_patch_args:nNNpn
\skip_vertical:c 10487 {
\skip_vertical:n 10488 {
10489 \__kernel_chk_expr:nNnN {#1} \etex_glueexpr:D { }
10490 \skip_horizontal:n
10491 }
10492 }
10493 \cs_new:Npn \skip_horizontal:n #1
10494 { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
10495 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
10496 \__kernel_patch_args:nNNpn
10497 {
10498 {
10499 \__kernel_chk_expr:nNnN {#1} \etex_glueexpr:D { }
10500 \skip_vertical:n
10501 }
10502 }
10503 \cs_new:Npn \skip_vertical:n #1
10504 { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
10505 \cs_generate_variant:Nn \skip_horizontal:N { c }
10506 \cs_generate_variant:Nn \skip_vertical:N { c }

```

(End definition for `\skip_horizontal:N` and others. These functions are documented on page 161.)

19.17 Viewing skip variables

```

\skip_show:N Diagnostics.
\skip_show:c 10507 \cs_new_eq:NN \skip_show:N \__kernel_register_show:N
10508 \cs_generate_variant:Nn \skip_show:N { c }

```

(End definition for `\skip_show:N`. This function is documented on page 160.)

`\skip_show:n` Diagnostics. We don't use the TeX primitive `\showthe` to show skip expressions: this gives a more unified output.

```

10509 \cs_new_protected:Npn \skip_show:n
10510 { \msg_show_eval:Nn \skip_eval:n }

```

(End definition for `\skip_show:n`. This function is documented on page 160.)

```

\skip_log:N Diagnostics. Redirect output of \skip_show:n to the log.
\skip_log:c 10511 \cs_new_eq:NN \skip_log:N \__kernel_register_log:N
\skip_log:n 10512 \cs_new_eq:NN \skip_log:c \__kernel_register_log:c
10513 \cs_new_protected:Npn \skip_log:n
10514 { \msg_log_eval:Nn \skip_eval:n }

```

(End definition for `\skip_log:N` and `\skip_log:n`. These functions are documented on page 161.)

19.18 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions but need to terminate correctly.

`\c_max_skip` 10515 `\skip_const:Nn \c_zero_skip { \c_zero_dim }`
 10516 `\skip_const:Nn \c_max_skip { \c_max_dim }`

(End definition for `\c_zero_skip` and `\c_max_skip`. These functions are documented on page 161.)

19.19 Scratch skips

`\l_tmpa_skip` We provide two local and two global scratch registers, maybe we need more or less.

`\l_tmpb_skip` 10517 `\skip_new:N \l_tmpa_skip`
`\g_tmpa_skip` 10518 `\skip_new:N \l_tmpb_skip`
`\g_tmpb_skip` 10519 `\skip_new:N \g_tmpa_skip`
 10520 `\skip_new:N \g_tmpb_skip`

(End definition for `\l_tmpa_skip` and others. These variables are documented on page 161.)

19.20 Creating and initialising muskip variables

`\muskip_new:N` And then we add muskips.

`\muskip_new:c` 10521 `{*package}`
 10522 `\cs_new_protected:Npn \muskip_new:N #1`
 10523 `{`
 10524 `__kernel_chk_if_free_cs:N #1`
 10525 `\cs:w newmuskip \cs_end: #1`
 10526 `}`
 10527 `{/package}`
 10528 `\cs_generate_variant:Nn \muskip_new:N { c }`

(End definition for `\muskip_new:N`. This function is documented on page 162.)

`\muskip_const:Nn` See `\skip_const:Nn`.

`\muskip_const:cn` 10529 `__kernel_patch:nnNNpn { __kernel_chk_var_scope:NN c #1 } { }`
 10530 `\cs_new_protected:Npn \muskip_const:Nn #1#2`
 10531 `{`
 10532 `\muskip_new:N #1`
 10533 `\tex_global:D #1 ~ \muskip_eval:n {#2} \scan_stop:`
 10534 `}`
 10535 `\cs_generate_variant:Nn \muskip_const:Nn { c }`

(End definition for `\muskip_const:Nn`. This function is documented on page 162.)

`\muskip_zero:N` Reset the register to zero.

`\muskip_zero:c` 10536 `__kernel_patch:nnNNpn { __kernel_chk_var_local:N #1 } { }`
`\muskip_gzero:N` 10537 `\cs_new_protected:Npn \muskip_zero:N #1`
`\muskip_gzero:c` 10538 `{ #1 \c_zero_muskip }`
 10539 `__kernel_patch:nnNNpn { __kernel_chk_var_global:N #1 } { }`
 10540 `\cs_new_protected:Npn \muskip_gzero:N #1`
 10541 `{ \tex_global:D #1 \c_zero_muskip }`
 10542 `\cs_generate_variant:Nn \muskip_zero:N { c }`
 10543 `\cs_generate_variant:Nn \muskip_gzero:N { c }`

(End definition for `\muskip_zero:N` and `\muskip_gzero:N`. These functions are documented on page 162.)

\muskip_zero_new:N Create a register if needed, otherwise clear it.

```

\muskip_zero_new:c 10544 \cs_new_protected:Npn \muskip_zero_new:N #1
\muskip_gzero_new:N 10545 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\muskip_gzero_new:c 10546 \cs_new_protected:Npn \muskip_gzero_new:N #1
10547 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
10548 \cs_generate_variant:Nn \muskip_zero_new:N { c }
10549 \cs_generate_variant:Nn \muskip_gzero_new:N { c }

(End definition for \muskip_zero_new:N and \muskip_gzero_new:N. These functions are documented on
page 162.)

```

\muskip_if_exist_p:N Copies of the cs functions defined in l3basics.

```

\muskip_if_exist_p:c 10550 \prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N
\muskip_if_exist:NTF 10551 { TF , T , F , p }
\muskip_if_exist:cTF 10552 \prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c
10553 { TF , T , F , p }

(End definition for \muskip_if_exist:NTF. This function is documented on page 162.)

```

19.21 Setting muskip variables

See skip case.

```

10554 \cs_set_protected:Npn \__skip_tmp:w #1#2#3
10555 {
10556   \__kernel_patch_args:nnnNpn
10557   { #1 ##1 }
10558   { }
10559   {
10560     {##1}
10561     {
10562       \__kernel_chk_expr:nNnN {##2}
10563       \etex_muexpr:D { \etex_mutoglue:D } #3
10564     }
10565   }
10566   #2 #3
10567 }

```

\muskip_set:Nn This should be pretty familiar.

```

\muskip_set:cn 10568 \__skip_tmp:w \__kernel_chk_var_local:N
\muskip_gset:Nn 10569 \cs_new_protected:Npn \muskip_set:Nn #1#2
\muskip_gset:cn 10570 { #1 ~ \etex_muexpr:D #2 \scan_stop: }
10571 \__skip_tmp:w \__kernel_chk_var_global:N
10572 \cs_new_protected:Npn \muskip_gset:Nn #1#2
10573 { \tex_global:D #1 ~ \etex_muexpr:D #2 \scan_stop: }
10574 \cs_generate_variant:Nn \muskip_set:Nn { c }
10575 \cs_generate_variant:Nn \muskip_gset:Nn { c }

(End definition for \muskip_set:Nn and \muskip_gset:Nn. These functions are documented on page
162.)

```

\muskip_set_eq:NN All straightforward.

```

\muskip_set_eq:cN 10576 \__kernel_patch:nnNpn { \__kernel_chk_var_local:N #1 } { }
\muskip_set_eq:Nc 10577 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\muskip_set_eq:cc 10578 \cs_generate_variant:Nn \muskip_set_eq:NN { c , Nc , cc }
\muskip_gset_eq:NN
\muskip_gset_eq:cN
\muskip_gset_eq:Nc
\muskip_gset_eq:cc

```

```

10579 \__kernel_patch:nNn { \__kernel_chk_var_global:N #1 } { }
10580 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
10581 \cs_generate_variant:Nn \muskip_gset_eq:NN { c , Nc , cc }

```

(End definition for `\muskip_set_eq:NN` and `\muskip_gset_eq:NN`. These functions are documented on page 162.)

```

\muskip_add:Nn Using by here deals with the (incorrect) case \muskip123.
\muskip_add:cn
\muskip_gadd:Nn
\muskip_gadd:cn
\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn
10582 \__skip_tmp:w \__kernel_chk_var_local:N
10583 \cs_new_protected:Npn \muskip_add:Nn #1#2
10584 { \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }
10585 \__skip_tmp:w \__kernel_chk_var_global:N
10586 \cs_new_protected:Npn \muskip_gadd:Nn #1#2
10587 { \tex_global:D \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }
10588 \cs_generate_variant:Nn \muskip_add:Nn { c }
10589 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
10590 \__skip_tmp:w \__kernel_chk_var_local:N
10591 \cs_new_protected:Npn \muskip_sub:Nn #1#2
10592 { \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }
10593 \__skip_tmp:w \__kernel_chk_var_global:N
10594 \cs_new_protected:Npn \muskip_gsub:Nn #1#2
10595 { \tex_global:D \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }
10596 \cs_generate_variant:Nn \muskip_sub:Nn { c }
10597 \cs_generate_variant:Nn \muskip_gsub:Nn { c }

```

(End definition for `\muskip_add:Nn` and others. These functions are documented on page 162.)

19.22 Using muskip expressions and variables

`\muskip_eval:n` Evaluating a muskip expression expandably.

```

10598 \__kernel_patch_args:nNn
10599 {
10600   {
10601     \__kernel_chk_expr:nNn {#1} \etex_muexpr:D
10602     { \etex_mutogluue:D } \muskip_eval:n
10603   }
10604 }
10605 \cs_new:Npn \muskip_eval:n #1
10606 { \muskip_use:N \etex_muexpr:D #1 \scan_stop: }

```

(End definition for `\muskip_eval:n`. This function is documented on page 163.)

`\muskip_use:N` Accessing a $\langle muskip \rangle$.

```

\muskip_use:c
10607 \cs_new_eq:NN \muskip_use:N \tex_the:D
10608 \cs_generate_variant:Nn \muskip_use:N { c }

```

(End definition for `\muskip_use:N`. This function is documented on page 163.)

19.23 Viewing muskip variables

`\muskip_show:N` Diagnostics.

`\muskip_show:c` 10609 `\cs_new_eq:NN \muskip_show:N __kernel_register_show:N`
 10610 `\cs_generate_variant:Nn \muskip_show:N { c }`

(End definition for `\muskip_show:N`. This function is documented on page 163.)

`\muskip_show:n` Diagnostics. We don't use the TeX primitive `\showthe` to show muskip expressions: this gives a more unified output.

10611 `\cs_new_protected:Npn \muskip_show:n`
 10612 `{ \msg_show_eval:Nn \muskip_eval:n }`

(End definition for `\muskip_show:n`. This function is documented on page 163.)

`\muskip_log:N` Diagnostics. Redirect output of `\muskip_show:n` to the log.

`\muskip_log:c` 10613 `\cs_new_eq:NN \muskip_log:N __kernel_register_log:N`
`\muskip_log:n` 10614 `\cs_new_eq:NN \muskip_log:c __kernel_register_log:c`
 10615 `\cs_new_protected:Npn \muskip_log:n`
 10616 `{ \msg_log_eval:Nn \muskip_eval:n }`

(End definition for `\muskip_log:N` and `\muskip_log:n`. These functions are documented on page 163.)

19.24 Constant muskips

`\c_zero_muskip` Constant muskips given by their value.

`\c_max_muskip` 10617 `\muskip_const:Nn \c_zero_muskip { 0 mu }`
 10618 `\muskip_const:Nn \c_max_muskip { 16383.99999 mu }`

(End definition for `\c_zero_muskip` and `\c_max_muskip`. These functions are documented on page 164.)

19.25 Scratch muskips

`\l_tmpa_muskip` We provide two local and two global scratch registers, maybe we need more or less.

`\l_tmpb_muskip` 10619 `\muskip_new:N \l_tmpa_muskip`
`\g_tmpa_muskip` 10620 `\muskip_new:N \l_tmpb_muskip`
`\g_tmpb_muskip` 10621 `\muskip_new:N \g_tmpa_muskip`
 10622 `\muskip_new:N \g_tmpb_muskip`

(End definition for `\l_tmpa_muskip` and others. These variables are documented on page 164.)

10623 `\</initex | package>`

20 l3keys Implementation

10624 `\<*initex | package>`

20.1 Low-level interface

The low-level key parser is based heavily on `keyval`, but with a number of additional “safety” requirements and with the idea that the parsed list of key–value pairs can be processed in a variety of ways. The net result is that this code needs around twice the amount of time as `keyval` to parse the same list of keys. To optimise speed as far as reasonably practical, a number of lower-level approaches are taken rather than using the higher-level `expl3` interfaces.

10625 `\<@@=keyval>`

`\l__keyval_key_tl` The current key name and value.
`\l__keyval_value_tl` 10626 `\tl_new:N \l__keyval_key_tl`
10627 `\tl_new:N \l__keyval_value_tl`
(End definition for \l__keyval_key_tl and \l__keyval_value_tl.)

`\l__keyval_sanitise_tl` A token list variable for dealing with awkward category codes in the input.
10628 `\tl_new:N \l__keyval_sanitise_tl`
(End definition for \l__keyval_sanitise_tl.)

`\keyval_parse:NNn` The main function starts off by normalising category codes in package mode. That’s relatively “expensive” so is skipped (hopefully) in format mode. We then hand off to the parser. The use of `\q_mark` here prevents loss of braces from the key argument. Notice that by passing the two processor commands along the input stack we avoid the need to track these at all.

```
10629 \cs_new_protected:Npn \keyval_parse:NNn #1#2#3
10630 {
10631   \*initex
10632   \__keyval_loop:NNw #1#2 \q_mark #3 , \q_recursion_tail ,
10633   \*initex
10634   \*package
10635   \tl_set:Nn \l__keyval_sanitise_tl {#3}
10636   \__keyval_sanitise_equals:
10637   \__keyval_sanitise_comma:
10638   \exp_after:wN \__keyval_loop:NNw \exp_after:wN #1 \exp_after:wN #2
10639   \exp_after:wN \q_mark \l__keyval_sanitise_tl , \q_recursion_tail ,
10640   \*package
10641 }
```

(End definition for \keyval_parse:NNn. This function is documented on page 177.)

`__keyval_sanitise_equals:` A reasonably fast search and replace set up specifically for the active tokens. The nature of the input is known so everything is hard-coded. With only two tokens to cover, the speed gain from using dedicated functions is worth it.

```
\__keyval_sanitise_comma:
  \__keyval_sanitise_equals_auxi:w
  \__keyval_sanitise_equals_auxii:w
  \__keyval_sanitise_comma_auxi:w
  \__keyval_sanitise_comma_auxii:w
  \__keyval_sanitise_aux:w
10642 \*package
10643 \group_begin:
10644   \char_set_catcode_active:n { '=' }
10645   \char_set_catcode_active:n { ',' }
10646   \cs_new_protected:Npn \__keyval_sanitise_equals:
10647   {
10648     \exp_after:wN \__keyval_sanitise_equals_auxi:w \l__keyval_sanitise_tl
10649     \q_mark = \q_nil =
10650     \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
10651   }
10652   \cs_new_protected:Npn \__keyval_sanitise_equals_auxi:w #1 =
10653   {
10654     \tl_set:Nn \l__keyval_sanitise_tl {#1}
10655     \__keyval_sanitise_equals_auxii:w
10656   }
10657   \cs_new_protected:Npn \__keyval_sanitise_equals_auxii:w #1 =
10658   {
10659     \if_meaning:w \q_nil #1 \scan_stop:
10660     \else:
```

```

10661         \tl_set:Nx \l__keyval_sanitise_tl
10662         {
10663             \exp_not:o \l__keyval_sanitise_tl
10664             \token_to_str:N =
10665             \exp_not:n {#1}
10666         }
10667         \exp_after:wN \__keyval_sanitise_equals_auxii:w
10668     \fi:
10669 }
10670 \cs_new_protected:Npn \__keyval_sanitise_comma:
10671 {
10672     \exp_after:wN \__keyval_sanitise_comma_auxi:w \l__keyval_sanitise_tl
10673     \q_mark , \q_nil ,
10674     \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
10675 }
10676 \cs_new_protected:Npn \__keyval_sanitise_comma_auxi:w #1 ,
10677 {
10678     \tl_set:Nn \l__keyval_sanitise_tl {#1}
10679     \__keyval_sanitise_comma_auxii:w
10680 }
10681 \cs_new_protected:Npn \__keyval_sanitise_comma_auxii:w #1 ,
10682 {
10683     \if_meaning:w \q_nil #1 \scan_stop:
10684     \else:
10685         \tl_set:Nx \l__keyval_sanitise_tl
10686         {
10687             \exp_not:o \l__keyval_sanitise_tl
10688             \token_to_str:N ,
10689             \exp_not:n {#1}
10690         }
10691         \exp_after:wN \__keyval_sanitise_comma_auxii:w
10692     \fi:
10693 }
10694 \group_end:
10695 \cs_new_protected:Npn \__keyval_sanitise_aux:w #1 \q_mark
10696 { \tl_set:Nn \l__keyval_sanitise_tl {#1} }
10697 </package>

```

(End definition for __keyval_sanitise_equals: and others.)

__keyval_loop:NNw A fast test for the end of the loop, remembering to remove the leading quark first. Assuming that is not the case, look for a key and value then loop around, re-inserting a leading quark in front of the next position.

```

10698 \cs_new_protected:Npn \__keyval_loop:NNw #1#2#3 ,
10699 {
10700     \exp_after:wN \if_meaning:w \exp_after:wN \q_recursion_tail
10701     \use_none:n #3 \prg_do_nothing:
10702     \else:
10703         \__keyval_split:NNw #1#2#3 == \q_stop
10704         \exp_after:wN \__keyval_loop:NNw \exp_after:wN #1 \exp_after:wN #2
10705         \exp_after:wN \q_mark
10706     \fi:
10707 }

```

(End definition for __keyval_loop:NNw.)

```

    \__keyval_split:NNw
\__keyval_split_value:NNw
    \__keyval_split_tidy:w
    \__keyval_action:

```

The value is picked up separately from the key so there can be another quark inserted at the front, keeping braces and allowing both parts to share the same code paths. The key is found first then there's a check that there is something there: this is biased to the common case of there actually being a key. For the value, we first need to see if there is anything to do: if there is, extract it. The appropriate action is then inserted in front of the key and value. Doing this using an assignment is marginally faster than an an expansion chain.

```

10708 \cs_new_protected:Npn \__keyval_split:NNw #1#2#3 =
10709 {
10710     \__keyval_def:Nn \l__keyval_key_tl {#3}
10711     \if_meaning:w \l__keyval_key_tl \c_empty_tl
10712         \exp_after:wN \__keyval_split_tidy:w
10713     \else:
10714         \exp_after:wN \__keyval_split_value:NNw
10715         \exp_after:wN #1
10716         \exp_after:wN #2
10717         \exp_after:wN \q_mark
10718     \fi:
10719 }
10720 \cs_new_protected:Npn \__keyval_split_value:NNw #1#2#3 = #4 \q_stop
10721 {
10722     \if:w \scan_stop: \tl_to_str:n {#4} \scan_stop:
10723         \cs_set:Npx \__keyval_action:
10724             { \exp_not:N #1 { \exp_not:o \l__keyval_key_tl } }
10725     \else:
10726         \if:w
10727             \scan_stop:
10728             \__kernel_tl_to_str:w \exp_after:wN { \use_none:n #4 }
10729             \scan_stop:
10730             \__keyval_def:Nn \l__keyval_value_tl {#3}
10731             \cs_set:Npx \__keyval_action:
10732                 {
10733                     \exp_not:N #2
10734                     { \exp_not:o \l__keyval_key_tl }
10735                     { \exp_not:o \l__keyval_value_tl }
10736                 }
10737         \else:
10738             \cs_set:Npn \__keyval_action:
10739                 {
10740                     \__kernel_msg_error:nn { kernel }
10741                     { misplaced-equals-sign }
10742                 }
10743         \fi:
10744     \fi:
10745     \__keyval_action:
10746 }
10747 \cs_new_protected:Npn \__keyval_split_tidy:w #1 \q_stop
10748 {
10749     \if:w
10750         \scan_stop:
10751         \__kernel_tl_to_str:w \exp_after:wN { \use_none:n #1 }
10752         \scan_stop:
10753     \else:

```

```

10754     \exp_after:wN \__keyval_empty_key:
10755     \fi:
10756   }
10757   \cs_new:Npn \__keyval_action: { }
10758   \cs_new_protected:Npn \__keyval_empty_key:
10759     { \__kernel_msg_error:nn { kernel } { misplaced-equals-sign } }

```

(End definition for `__keyval_split:NNw` and others.)

```

\__keyval_def:Nn First remove the leading quark, then trim spaces off, and finally remove a set of braces.
\__keyval_def_aux:n
\__keyval_def_aux:w
10760 \cs_new_protected:Npn \__keyval_def:Nn #1#2
10761   {
10762     \tl_set:Nx #1
10763       { \tl_trim_spaces_apply:oN { \use_none:n #2 } \__keyval_def_aux:n }
10764   }
10765   \cs_new:Npn \__keyval_def_aux:n #1
10766     { \__keyval_def_aux:w #1 \q_stop }
10767   \cs_new:Npn \__keyval_def_aux:w #1 \q_stop { \exp_not:n {#1} }

```

(End definition for `__keyval_def:Nn`, `__keyval_def_aux:n`, and `__keyval_def_aux:w`.)

One message for the low level parsing system.

```

10768 \__kernel_msg_new:nnnn { kernel } { misplaced-equals-sign }
10769   { Misplaced-equals-sign~in~key-value~input~\msg_line_number: }
10770   {
10771     LaTeX-is-attempting-to-parse-some-key-value-input-but-found~
10772     two~equals~signs~not~separated~by~a~comma.
10773   }

```

20.2 Constants and variables

```

10774 <@@=keys>

```

Various storage areas for the different data which make up keys.

```

\c__keys_code_root_tl
\c__keys_default_root_tl
\c__keys_groups_root_tl
\c__keys_inherit_root_tl
\c__keys_type_root_tl
\c__keys_validate_root_tl
10775 \tl_const:Nn \c__keys_code_root_tl { key-code~>~ }
10776 \tl_const:Nn \c__keys_default_root_tl { key-default~>~ }
10777 \tl_const:Nn \c__keys_groups_root_tl { key-groups~>~ }
10778 \tl_const:Nn \c__keys_inherit_root_tl { key-inherit~>~ }
10779 \tl_const:Nn \c__keys_type_root_tl { key-type~>~ }
10780 \tl_const:Nn \c__keys_validate_root_tl { key-validate~>~ }

```

(End definition for `\c__keys_code_root_tl` and others.)

`\c__keys_props_root_tl` The prefix for storing properties.

```

10781 \tl_const:Nn \c__keys_props_root_tl { key-prop~>~ }

```

(End definition for `\c__keys_props_root_tl`.)

`\l_keys_choice_int` Publicly accessible data on which choice is being used when several are generated as a set.

```

10782 \int_new:N \l_keys_choice_int
10783 \tl_new:N \l_keys_choice_tl

```

(End definition for `\l_keys_choice_int` and `\l_keys_choice_tl`. These variables are documented on page 171.)

`\l__keys_groups_clist` Used for storing and recovering the list of groups which apply to a key: set as a comma list but at one point we have to use this for a token list recovery.

```
10784 \clist_new:N \l__keys_groups_clist
```

(End definition for \l__keys_groups_clist.)

`\l_keys_key_tl` The name of a key itself: needed when setting keys.

```
10785 \tl_new:N \l_keys_key_tl
```

(End definition for \l_keys_key_tl. This variable is documented on page 173.)

`\l__keys_module_tl` The module for an entire set of keys.

```
10786 \tl_new:N \l__keys_module_tl
```

(End definition for \l__keys_module_tl.)

`\l__keys_no_value_bool` A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.

```
10787 \bool_new:N \l__keys_no_value_bool
```

(End definition for \l__keys_no_value_bool.)

`\l__keys_only_known_bool` Used to track if only “known” keys are being set.

```
10788 \bool_new:N \l__keys_only_known_bool
```

(End definition for \l__keys_only_known_bool.)

`\l_keys_path_tl` The “path” of the current key is stored here: this is available to the programmer and so is public.

```
10789 \tl_new:N \l_keys_path_tl
```

(End definition for \l_keys_path_tl. This variable is documented on page 173.)

`\l__keys_property_tl` The “property” begin set for a key at definition time is stored here.

```
10790 \tl_new:N \l__keys_property_tl
```

(End definition for \l__keys_property_tl.)

`\l__keys_selective_bool` Two flags for using key groups: one to indicate that “selective” setting is active, a second to specify which type (“opt-in” or “opt-out”).

```
10791 \bool_new:N \l__keys_selective_bool
```

```
10792 \bool_new:N \l__keys_filtered_bool
```

(End definition for \l__keys_selective_bool and \l__keys_filtered_bool.)

`\l__keys_selective_seq` The list of key groups being filtered in or out during selective setting.

```
10793 \seq_new:N \l__keys_selective_seq
```

(End definition for \l__keys_selective_seq.)

`\l__keys_unused_clist` Used when setting only some keys to store those left over.

```
10794 \tl_new:N \l__keys_unused_clist
```

(End definition for \l__keys_unused_clist.)

`\l_keys_value_tl` The value given for a key: may be empty if no value was given.
10795 `\tl_new:N \l_keys_value_tl`
(End definition for `\l_keys_value_tl`. This variable is documented on page 173.)

`\l__keys_tmp_bool` Scratch space.
10796 `\bool_new:N \l__keys_tmp_bool`
(End definition for `\l__keys_tmp_bool`.)

20.3 The key defining mechanism

`\keys_define:nn` The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

```

10797 \cs_new_protected:Npn \keys_define:nn
10798 { \__keys_define:onn \l__keys_module_tl }
10799 \cs_new_protected:Npn \__keys_define:nnn #1#2#3
10800 {
10801   \tl_set:Nx \l__keys_module_tl { \__keys_remove_spaces:n {#2} }
10802   \keyval_parse:NNn \__keys_define:n \__keys_define:nn {#3}
10803   \tl_set:Nn \l__keys_module_tl {#1}
10804 }
10805 \cs_generate_variant:Nn \__keys_define:nnn { o }

```

(End definition for `\keys_define:nn` and `__keys_define:nnn`. This function is documented on page 166.)

`__keys_define:n`
`__keys_define:nn`
`__keys_define_aux:nn` The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```

10806 \cs_new_protected:Npn \__keys_define:n #1
10807 {
10808   \bool_set_true:N \l__keys_no_value_bool
10809   \__keys_define_aux:nn {#1} { }
10810 }
10811 \cs_new_protected:Npn \__keys_define:nn #1#2
10812 {
10813   \bool_set_false:N \l__keys_no_value_bool
10814   \__keys_define_aux:nn {#1} {#2}
10815 }
10816 \cs_new_protected:Npn \__keys_define_aux:nn #1#2
10817 {
10818   \__keys_property_find:n {#1}
10819   \cs_if_exist:cTF { \c__keys_props_root_tl \l__keys_property_tl }
10820   { \__keys_define_code:n {#2} }
10821   {
10822     \tl_if_empty:NF \l__keys_property_tl
10823     {
10824       \__kernel_msg_error:nxxx { kernel } { key-property-unknown }
10825       { \l__keys_property_tl } { \l_keys_path_tl }
10826     }
10827   }
10828 }

```

(End definition for `_keys_define:n`, `_keys_define:nn`, and `_keys_define_aux:nn`.)

`_keys_property_find:n` Searching for a property means finding the last . in the input, and storing the text before and after it. Everything is turned into strings, so there is no problem using an x-type expansion.

`_keys_property_find:w`

```

10829 \cs_new_protected:Npn \_keys_property_find:n #1
10830 {
10831   \tl_set:Nx \l__keys_property_tl { \_keys_remove_spaces:n {#1} }
10832   \exp_after:wN \_keys_property_find:w \l__keys_property_tl . .
10833   \q_stop {#1}
10834 }
10835 \cs_new_protected:Npn \_keys_property_find:w #1 . #2 . #3 \q_stop #4
10836 {
10837   \tl_if_blank:nTF {#3}
10838   {
10839     \tl_clear:N \l__keys_property_tl
10840     \__kernel_msg_error:nnn { kernel } { key-no-property } {#4}
10841   }
10842   {
10843     \str_if_eq:nnTF {#3} { . }
10844     {
10845       \tl_set:Nx \l_keys_path_tl
10846       {
10847         \tl_if_empty:NF \l__keys_module_tl
10848         { \l__keys_module_tl / }
10849         #1
10850       }
10851       \tl_set:Nn \l__keys_property_tl { . #2 }
10852     }
10853     {
10854       \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / #1 . #2 }
10855       \_keys_property_search:w #3 \q_stop
10856     }
10857   }
10858 }
10859 \cs_new_protected:Npn \_keys_property_search:w #1 . #2 \q_stop
10860 {
10861   \str_if_eq:nnTF {#2} { . }
10862   {
10863     \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl }
10864     \tl_set:Nn \l__keys_property_tl { . #1 }
10865   }
10866   {
10867     \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . #1 }
10868     \_keys_property_search:w #2 \q_stop
10869   }
10870 }

```

(End definition for `_keys_property_find:n` and `_keys_property_find:w`.)

`_keys_define_code:n`

`_keys_define_code:w`

Two possible cases. If there is a value for the key, then just use the function. If not, then a check to make sure there is no need for a value with the property. If there should be one then complain, otherwise execute it. There is no need to check for a : as if it was missing the earlier tests would have failed.

```

10871 \cs_new_protected:Npn \__keys_define_code:n #1
10872 {
10873   \bool_if:NTF \l__keys_no_value_bool
10874   {
10875     \exp_after:wN \__keys_define_code:w
10876     \l__keys_property_tl \q_stop
10877     { \use:c { \c__keys_props_root_tl \l__keys_property_tl } }
10878     {
10879       \__kernel_msg_error:nxx { kernel }
10880       { key-property-requires-value } { \l__keys_property_tl }
10881       { \l_keys_path_tl }
10882     }
10883   }
10884   { \use:c { \c__keys_props_root_tl \l__keys_property_tl } {#1} }
10885 }
10886 \exp_last_unbraced:NNNNo
10887 \cs_new:Npn \__keys_define_code:w #1 \c_colon_str #2 \q_stop
10888 { \tl_if_empty:nTF {#2} }

```

(End definition for __keys_define_code:n and __keys_define_code:w.)

20.4 Turning properties into actions

__keys_bool_set:Nn Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```

10889 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
10890 {
10891   \bool_if_exist:NF #1 { \bool_new:N #1 }
10892   \__keys_choice_make:
10893   \__keys_cmd_set:nx { \l_keys_path_tl / true }
10894   { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
10895   \__keys_cmd_set:nx { \l_keys_path_tl / false }
10896   { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
10897   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
10898   {
10899     \__kernel_msg_error:nxx { kernel } { boolean-values-only }
10900     { \l_keys_key_tl }
10901   }
10902   \__keys_default_set:n { true }
10903 }
10904 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }

```

(End definition for __keys_bool_set:Nn.)

__keys_bool_set_inverse:Nn Inverse boolean setting is much the same.

```

10905 \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
10906 {
10907   \bool_if_exist:NF #1 { \bool_new:N #1 }
10908   \__keys_choice_make:
10909   \__keys_cmd_set:nx { \l_keys_path_tl / true }
10910   { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
10911   \__keys_cmd_set:nx { \l_keys_path_tl / false }
10912   { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
10913   \__keys_cmd_set:nn { \l_keys_path_tl / unknown }

```

```

10914     {
10915         \_kernel_msg_error:nxx { kernel } { boolean-values-only }
10916         { \l_keys_key_tl }
10917     }
10918     \_keys_default_set:n { true }
10919 }
10920 \cs_generate_variant:Nn \_keys_bool_set_inverse:Nn { c }

```

(End definition for _keys_bool_set_inverse:Nn.)

_keys_choice_make: To make a choice from a key, two steps: set the code, and set the unknown key. As
_keys_multichoice_make: multichoice and choices are essentially the same bar one function, the code is given
_keys_choice_make:N together.
_keys_choice_make_aux:N

```

10921 \cs_new_protected:Npn \_keys_choice_make:
10922 { \_keys_choice_make:N \_keys_choice_find:n }
10923 \cs_new_protected:Npn \_keys_multichoice_make:
10924 { \_keys_choice_make:N \_keys_multichoice_find:n }
10925 \cs_new_protected:Npn \_keys_choice_make:N #1
10926 {
10927     \cs_if_exist:cTF
10928     { \c__keys_type_root_tl \_keys_parent:o \l_keys_path_tl }
10929     {
10930         \str_if_eq_x:nnTF
10931         {
10932             \exp_not:v
10933             { \c__keys_type_root_tl \_keys_parent:o \l_keys_path_tl }
10934         }
10935         { choice }
10936         {
10937             \_kernel_msg_error:nxxx { kernel } { nested-choice-key }
10938             { \l_keys_path_tl } { \_keys_parent:o \l_keys_path_tl }
10939         }
10940         { \_keys_choice_make_aux:N #1 }
10941     }
10942     { \_keys_choice_make_aux:N #1 }
10943 }
10944 \cs_new_protected:Npn \_keys_choice_make_aux:N #1
10945 {
10946     \cs_set_nopar:cpn { \c__keys_type_root_tl \l_keys_path_tl }
10947     { choice }
10948     \_keys_cmd_set:nn { \l_keys_path_tl } { #1 {##1} }
10949     \_keys_cmd_set:nn { \l_keys_path_tl / unknown }
10950     {
10951         \_kernel_msg_error:nxxx { kernel } { key-choice-unknown }
10952         { \l_keys_path_tl } {##1}
10953     }
10954 }

```

(End definition for _keys_choice_make: and others.)

_keys_choices_make:nn Auto-generating choices means setting up the root key as a choice, then defining each
_keys_multichoice_make:nn choice in turn.
_keys_choices_make:Nnn

```

10955 \cs_new_protected:Npn \_keys_choices_make:nn
10956 { \_keys_choices_make:Nnn \_keys_choice_make: }

```

```

10957 \cs_new_protected:Npn \__keys_multichoices_make:nn
10958 { \__keys_choices_make:Nnn \__keys_multichoice_make: }
10959 \cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
10960 {
10961   #1
10962   \int_zero:N \l_keys_choice_int
10963   \clist_map_inline:nn {#2}
10964   {
10965     \int_incr:N \l_keys_choice_int
10966     \__keys_cmd_set:nx
10967     { \l_keys_path_tl / \__keys_remove_spaces:n {##1} }
10968     {
10969       \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
10970       \int_set:Nn \exp_not:N \l_keys_choice_int
10971       { \int_use:N \l_keys_choice_int }
10972       \exp_not:n {#3}
10973     }
10974   }
10975 }

```

(End definition for __keys_choices_make:nn, __keys_multichoices_make:nn, and __keys_choices_make:Nnn.)

__keys_cmd_set:nn Setting the code for a key first logs if appropriate that we are defining a new key, then
 __keys_cmd_set:nx saves the code.

```

\__keys_cmd_set:Vn
\__keys_cmd_set:Vo
10976 \__kernel_patch:nnNpn
10977 {
10978   \cs_if_exist:cF { \c__keys_code_root_tl #1 }
10979   { \__kernel_debug_log:x { Defining~key~#1~\msg_line_context: } }
10980 }
10981 { }
10982 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
10983 { \cs_set_protected:cpn { \c__keys_code_root_tl #1 } ##1 {#2} }
10984 \cs_generate_variant:Nn \__keys_cmd_set:nn { nx , Vn , Vo }

```

(End definition for __keys_cmd_set:nn.)

__keys_default_set:n Setting a default value is easy. These are stored using \cs_set:cpx as this avoids any
 worries about whether a token list exists.

```

10985 \cs_new_protected:Npn \__keys_default_set:n #1
10986 {
10987   \tl_if_empty:nTF {#1}
10988   {
10989     \cs_set_eq:cN
10990     { \c__keys_default_root_tl \l_keys_path_tl }
10991     \tex_undefined:D
10992   }
10993   {
10994     \cs_set:cpx
10995     { \c__keys_default_root_tl \l_keys_path_tl }
10996     { \exp_not:n {#1} }
10997   }
10998 }

```

(End definition for __keys_default_set:n.)

`__keys_groups_set:n` Assigning a key to one or more groups uses comma lists. As the list of groups only exists if there is anything to do, the setting is done using a scratch list. For the usual grouping reasons we use the low-level approach to undefining a list. We also use the low-level approach for the other case to avoid tripping up the `check-declarations` code.

```

10999 \cs_new_protected:Npn \__keys_groups_set:n #1
11000 {
11001   \clist_set:Nn \l__keys_groups_clist {#1}
11002   \clist_if_empty:NTF \l__keys_groups_clist
11003   {
11004     \cs_set_eq:cN { \c__keys_groups_root_tl \l_keys_path_tl }
11005     \tex_undefined:D
11006   }
11007   {
11008     \cs_set_eq:cN { \c__keys_groups_root_tl \l_keys_path_tl }
11009     \l__keys_groups_clist
11010   }
11011 }

```

(End definition for __keys_groups_set:n.)

`__keys_inherit:n` Inheritance means ignoring anything already said about the key: zap the lot and set up.

```

11012 \cs_new_protected:Npn \__keys_inherit:n #1
11013 {
11014   \__keys_undefine:
11015   \cs_set_nopar:cpn { \c__keys_inherit_root_tl \l_keys_path_tl } {#1}
11016 }

```

(End definition for __keys_inherit:n.)

`__keys_initialise:n` A set up for initialisation: just run the code if it exists.

```

11017 \cs_new_protected:Npn \__keys_initialise:n #1
11018 {
11019   \cs_if_exist_use:cT { \c__keys_code_root_tl \l_keys_path_tl } { {#1} }
11020 }

```

(End definition for __keys_initialise:n.)

`__keys_meta_make:n` To create a meta-key, simply set up to pass data through.

```

\__keys_meta_make:nn
11021 \cs_new_protected:Npn \__keys_meta_make:n #1
11022 {
11023   \__keys_cmd_set:Vo \l_keys_path_tl
11024   {
11025     \exp_after:wN \keys_set:nn
11026     \exp_after:wN { \l__keys_module_tl } {#1}
11027   }
11028 }
11029 \cs_new_protected:Npn \__keys_meta_make:nn #1#2
11030 { \__keys_cmd_set:Vn \l_keys_path_tl { \keys_set:nn {#1} {#2} } }

```

(End definition for __keys_meta_make:n and __keys_meta_make:nn.)

`__keys_undefine:` `U`ndefining a key has to be done without `\cs_undefine:c` as that function acts globally.

```

11031 \cs_new_protected:Npn \__keys_undefine:
11032 {
11033   \clist_map_inline:nn
11034     { code , default , groups , inherit , type , validate }
11035     {
11036       \cs_set_eq:cN
11037         { \tl_use:c { c__keys_ ##1 _root_tl } \l_keys_path_tl }
11038         \tex_undefined:D
11039     }
11040 }
```

(End definition for __keys_undefine:.)

`__keys_value_requirement:nn` Validating key input is done using a second function which runs before the main key code. Setting that up means setting it equal to a generic stub which does the check. This approach makes the lookup very fast at the cost of one additional csname per key that needs it. The cleanup here has to know the structure of the following code.

`__keys_validate_forbidden:`

`__keys_validate_required:`

`__keys_validate_cleanup:w`

```

11041 \cs_new_protected:Npn \__keys_value_requirement:nn #1#2
11042 {
11043   \str_case:nnF {#2}
11044   {
11045     { true }
11046     {
11047       \cs_set_eq:cc
11048         { \c__keys_validate_root_tl \l_keys_path_tl }
11049         { __keys_validate_ #1 : }
11050     }
11051     { false }
11052     {
11053       \cs_if_eq:ccT
11054         { \c__keys_validate_root_tl \l_keys_path_tl }
11055         { __keys_validate_ #1 : }
11056         {
11057           \cs_set_eq:cN
11058             { \c__keys_validate_root_tl \l_keys_path_tl }
11059             \tex_undefined:D
11060         }
11061     }
11062   }
11063   {
11064     \__kernel_msg_error:nxx { kernel }
11065     { key-property-boolean-values-only }
11066     { .value_ #1 :n }
11067   }
11068 }
11069 \cs_new_protected:Npn \__keys_validate_forbidden:
11070 {
11071   \bool_if:NF \l__keys_no_value_bool
11072   {
11073     \__kernel_msg_error:nxx { kernel } { value-forbidden }
11074     { \l_keys_path_tl } { \l_keys_value_tl }
11075     \__keys_validate_cleanup:w
11076   }
```



```

11077     }
11078 \cs_new_protected:Npn \__keys_validate_required:
11079 {
11080     \bool_if:NT \l__keys_no_value_bool
11081     {
11082         \__kernel_msg_error:nnx { kernel } { value-required }
11083         { \l_keys_path_tl }
11084         \__keys_validate_cleanup:w
11085     }
11086 }
11087 \cs_new_protected:Npn \__keys_validate_cleanup:w #1 \cs_end: #2#3 { }

```

(End definition for __keys_value_requirement:nn and others.)

__keys_variable_set:NnnN Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

__keys_variable_set:cnnN

```

11088 \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
11089 {
11090     \use:c { #2_if_exist:NF } #1 { \use:c { #2_new:N } #1 }
11091     \__keys_cmd_set:nx { \l_keys_path_tl }
11092     {
11093         \exp_not:c { #2 _ #3 set:N #4 }
11094         \exp_not:N #1
11095         \exp_not:n { {##1} }
11096     }
11097 }
11098 \cs_generate_variant:Nn \__keys_variable_set:NnnN { c }

```

(End definition for __keys_variable_set:NnnN.)

20.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have “normal” argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

```

. bool_set:N One function for this.
. bool_set:c 11099 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:N } #1
. bool_gset:N 11100 { \__keys_bool_set:Nn #1 { } }
. bool_gset:c 11101 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:c } #1
11102 { \__keys_bool_set:cn {#1} { } }
11103 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:N } #1
11104 { \__keys_bool_set:Nn #1 { g } }
11105 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:c } #1
11106 { \__keys_bool_set:cn {#1} { g } }

```

(End definition for .bool_set:N and .bool_gset:N. These functions are documented on page 167.)

```

. bool_set_inverse:N One function for this.
. bool_set_inverse:c 11107 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:N } #1
. bool_gset_inverse:N 11108 { \__keys_bool_set_inverse:Nn #1 { } }
. bool_gset_inverse:c

```

```

11109 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:c } #1
11110 { \__keys_bool_set_inverse:cn {#1} { } }
11111 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:N } #1
11112 { \__keys_bool_set_inverse:Nn #1 { g } }
11113 \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:c } #1
11114 { \__keys_bool_set_inverse:cn {#1} { g } }

```

(End definition for `.bool_set_inverse:N` and `.bool_gset_inverse:N`. These functions are documented on page 167.)

.choice: Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```

11115 \cs_new_protected:cpn { \c__keys_props_root_tl .choice: }
11116 { \__keys_choice_make: }

```

(End definition for `.choice:`. This function is documented on page 167.)

.choices:nn For auto-generation of a series of mutually-exclusive choices. Here, `#1` consists of two separate arguments, hence the slightly odd-looking implementation.

```

11117 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:nn } #1
11118 { \__keys_choices_make:nn #1 }
11119 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:Vn } #1
11120 { \exp_args:NV \__keys_choices_make:nn #1 }
11121 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:on } #1
11122 { \exp_args:No \__keys_choices_make:nn #1 }
11123 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:xn } #1
11124 { \exp_args:Nx \__keys_choices_make:nn #1 }

```

(End definition for `.choices:nn`. This function is documented on page 167.)

.code:n Creating code is simply a case of passing through to the underlying `set` function.

```

11125 \cs_new_protected:cpn { \c__keys_props_root_tl .code:n } #1
11126 { \__keys_cmd_set:nn { \l_keys_path_tl } {#1} }

```

(End definition for `.code:n`. This function is documented on page 167.)

.clist_set:N

```

11127 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:N } #1
11128 { \__keys_variable_set:NnnN #1 { clist } { } n }
11129 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:c } #1
11130 { \__keys_variable_set:cnnN {#1} { clist } { } n }
11131 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:N } #1
11132 { \__keys_variable_set:NnnN #1 { clist } { g } n }
11133 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:c } #1
11134 { \__keys_variable_set:cnnN {#1} { clist } { g } n }

```

(End definition for `.clist_set:N` and `.clist_gset:N`. These functions are documented on page 167.)

.default:n Expansion is left to the internal functions.

```

11135 \cs_new_protected:cpn { \c__keys_props_root_tl .default:n } #1
11136 { \__keys_default_set:n {#1} }
11137 \cs_new_protected:cpn { \c__keys_props_root_tl .default:V } #1
11138 { \exp_args:NV \__keys_default_set:n #1 }
11139 \cs_new_protected:cpn { \c__keys_props_root_tl .default:o } #1
11140 { \exp_args:No \__keys_default_set:n {#1} }
11141 \cs_new_protected:cpn { \c__keys_props_root_tl .default:x } #1
11142 { \exp_args:Nx \__keys_default_set:n {#1} }

```

(End definition for `.default:n`. This function is documented on page 168.)

```
.dim_set:N Setting a variable is very easy: just pass the data along.
.dim_set:c 11143 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:N } #1
           { \__keys_variable_set:NnnN #1 { dim } { } n }
.dim_gset:N 11144 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:c } #1
           { \__keys_variable_set:cnnN {#1} { dim } { } n }
.dim_gset:c 11145 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1
           { \__keys_variable_set:NnnN #1 { dim } { g } n }
           11146 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
           { \__keys_variable_set:cnnN {#1} { dim } { g } n }
           11147 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1
           { \__keys_variable_set:NnnN #1 { dim } { g } n }
           11148 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
           { \__keys_variable_set:cnnN {#1} { dim } { g } n }
           11149 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
           { \__keys_variable_set:cnnN {#1} { dim } { g } n }
           11150
```

(End definition for `.dim_set:N` and `.dim_gset:N`. These functions are documented on page 168.)

```
.fp_set:N Setting a variable is very easy: just pass the data along.
.fp_set:c 11151 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:N } #1
           { \__keys_variable_set:NnnN #1 { fp } { } n }
.fp_gset:N 11152 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:c } #1
           { \__keys_variable_set:cnnN {#1} { fp } { } n }
.fp_gset:c 11153 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1
           { \__keys_variable_set:NnnN #1 { fp } { g } n }
           11154 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1
           { \__keys_variable_set:cnnN {#1} { fp } { g } n }
           11155 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
           { \__keys_variable_set:NnnN #1 { fp } { g } n }
           11156 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
           { \__keys_variable_set:cnnN {#1} { fp } { g } n }
           11157 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
           { \__keys_variable_set:cnnN {#1} { fp } { g } n }
           11158
```

(End definition for `.fp_set:N` and `.fp_gset:N`. These functions are documented on page 168.)

```
.groups:n A single property to create groups of keys.
           11159 \cs_new_protected:cpn { \c__keys_props_root_tl .groups:n } #1
           { \__keys_groups_set:n {#1} }
```

(End definition for `.groups:n`. This function is documented on page 168.)

```
.inherit:n Nothing complex: only one variant at the moment!
           11161 \cs_new_protected:cpn { \c__keys_props_root_tl .inherit:n } #1
           { \__keys_inherit:n {#1} }
```

(End definition for `.inherit:n`. This function is documented on page 168.)

```
.initial:n The standard hand-off approach.
.initial:V 11163 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:n } #1
.initial:o 11164 { \__keys_initialise:n {#1} }
.initial:x 11165 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:V } #1
           { \exp_args:NV \__keys_initialise:n #1 }
           11166 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:o } #1
           { \exp_args:No \__keys_initialise:n {#1} }
           11167 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
           { \exp_args:Nx \__keys_initialise:n {#1} }
           11168 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
           { \exp_args:Nx \__keys_initialise:n {#1} }
           11169 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
           { \exp_args:Nx \__keys_initialise:n {#1} }
           11170
```

(End definition for `.initial:n`. This function is documented on page 169.)

.int_set:N Setting a variable is very easy: just pass the data along.

```

.int_set:c 11171 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:N } #1
.int_gset:N 11172 { \__keys_variable_set:NnnN #1 { int } { } n }
.int_gset:c 11173 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:c } #1
11174 { \__keys_variable_set:cnnN {#1} { int } { } n }
11175 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
11176 { \__keys_variable_set:NnnN #1 { int } { g } n }
11177 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
11178 { \__keys_variable_set:cnnN {#1} { int } { g } n }

```

(End definition for .int_set:N and .int_gset:N. These functions are documented on page 169.)

.meta:n Making a meta is handled internally.

```

11179 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:n } #1
11180 { \__keys_meta_make:n {#1} }

```

(End definition for .meta:n. This function is documented on page 169.)

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.

```

11181 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:nn } #1
11182 { \__keys_meta_make:nn #1 }

```

(End definition for .meta:nn. This function is documented on page 169.)

.multichoice: The same idea as .choice: and .choices:nn, but where more than one choice is allowed.

.multichoices:nn

```

11183 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoice: }
11184 { \__keys_multichoice_make: }
.multichoices:Vn 11185 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:nn } #1
11186 { \__keys_multichoices_make:nn #1 }
.multichoices:on 11187 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:Vn } #1
11188 { \exp_args:NV \__keys_multichoices_make:nn #1 }
11189 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:on } #1
11190 { \exp_args:No \__keys_multichoices_make:nn #1 }
11191 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:xn } #1
11192 { \exp_args:Nx \__keys_multichoices_make:nn #1 }

```

(End definition for .multichoice: and .multichoices:nn. These functions are documented on page 169.)

.skip_set:N Setting a variable is very easy: just pass the data along.

```

.skip_set:c 11193 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:N } #1
11194 { \__keys_variable_set:NnnN #1 { skip } { } n }
.skip_gset:N 11195 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:c } #1
11196 { \__keys_variable_set:cnnN {#1} { skip } { } n }
11197 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:N } #1
11198 { \__keys_variable_set:NnnN #1 { skip } { g } n }
11199 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:c } #1
11200 { \__keys_variable_set:cnnN {#1} { skip } { g } n }

```

(End definition for .skip_set:N and .skip_gset:N. These functions are documented on page 169.)

```

.tl_set:N Setting a variable is very easy: just pass the data along.
.tl_set:c 11201 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:N } #1
.tl_gset:N 11202 { \__keys_variable_set:NnnN #1 { tl } { } n }
.tl_gset:c 11203 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:c } #1
.tl_set_x:N 11204 { \__keys_variable_set:cnnN {#1} { tl } { } n }
.tl_set_x:c 11205 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:N } #1
.tl_gset_x:N 11206 { \__keys_variable_set:NnnN #1 { tl } { } x }
.tl_gset_x:c 11207 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:c } #1
11208 { \__keys_variable_set:cnnN {#1} { tl } { } x }
11209 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:N } #1
11210 { \__keys_variable_set:NnnN #1 { tl } { g } n }
11211 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:c } #1
11212 { \__keys_variable_set:cnnN {#1} { tl } { g } n }
11213 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:N } #1
11214 { \__keys_variable_set:NnnN #1 { tl } { g } x }
11215 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:c } #1
11216 { \__keys_variable_set:cnnN {#1} { tl } { g } x }

```

(End definition for .tl_set:N and others. These functions are documented on page 169.)

.undefine: Another simple wrapper.

```

11217 \cs_new_protected:cpn { \c__keys_props_root_tl .undefine: }
11218 { \__keys_undefine: }

```

(End definition for .undefine:. This function is documented on page 170.)

.value_forbidden:n These are very similar, so both call the same function.

```

.value_required:n 11219 \cs_new_protected:cpn { \c__keys_props_root_tl .value_forbidden:n } #1
11220 { \__keys_value_requirement:nn { forbidden } {#1} }
11221 \cs_new_protected:cpn { \c__keys_props_root_tl .value_required:n } #1
11222 { \__keys_value_requirement:nn { required } {#1} }

```

(End definition for .value_forbidden:n and .value_required:n. These functions are documented on page 170.)

20.6 Setting keys

\keys_set:nn A simple wrapper again.

```

\keys_set:nV 11223 \cs_new_protected:Npn \keys_set:nn
\keys_set:nv 11224 { \__keys_set:onnn { \l__keys_module_tl } }
\keys_set:no 11225 \cs_new_protected:Npn \__keys_set:nnn #1#2#3
\__keys_set:nnn 11226 {
\__keys_set:onnn 11227 \tl_set:Nx \l__keys_module_tl { \__keys_remove_spaces:n {#2} }
11228 \keyval_parse:NNn \__keys_set:n \__keys_set:nn {#3}
11229 \tl_set:Nn \l__keys_module_tl {#1}
11230 }
11231 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
11232 \cs_generate_variant:Nn \__keys_set:nnn { o }

```

(End definition for \keys_set:nn and __keys_set:nnn. This function is documented on page 173.)

`\keys_set_known:nnN` Setting known keys simply means setting the appropriate flag, then running the standard code. To allow for nested setting, any existing value of `\l__keys_unused_clist` is saved on the stack and reset afterwards. Note that for speed/simplicity reasons we use a `tl` operation to set the `clist` here!

```

\keys_set_known:nnN
\keys_set_known:nVN
\keys_set_known:nvN
\keys_set_known:noN
\__keys_set_known:nnnN
\__keys_set_known:onnN
\keys_set_known:nn
\keys_set_known:nV
\keys_set_known:nv
\keys_set_known:no
\__keys_keys_set_known:nn
11233 \cs_new_protected:Npn \keys_set_known:nnN
11234 { \__keys_set_known:onnN \l__keys_unused_clist }
11235 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
11236 \cs_new_protected:Npn \__keys_set_known:nnnN #1#2#3#4
11237 {
11238   \clist_clear:N \l__keys_unused_clist
11239   \keys_set_known:nn {#2} {#3}
11240   \tl_set:Nx #4 { \exp_not:o { \l__keys_unused_clist } }
11241   \tl_set:Nn \l__keys_unused_clist {#1}
11242 }
11243 \cs_generate_variant:Nn \__keys_set_known:nnnN { o }
11244 \cs_new_protected:Npn \keys_set_known:nn #1#2
11245 {
11246   \bool_if:NTF \l__keys_only_known_bool
11247   { \keys_set:nn }
11248   { \__keys_set_known:nn }
11249   {#1} {#2}
11250 }
11251 \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }
11252 \cs_new_protected:Npn \__keys_set_known:nn #1#2
11253 {
11254   \bool_set_true:N \l__keys_only_known_bool
11255   \keys_set:nn {#1} {#2}
11256   \bool_set_false:N \l__keys_only_known_bool
11257 }

```

(End definition for `\keys_set_known:nnN` and others. These functions are documented on page 174.)

`\keys_set_filter:nnnN` The idea of setting keys in a selective manner again uses flags wrapped around the basic code. The comments on `\keys_set_known:nnN` also apply here. We have a bit more shuffling to do to keep everything nestable.

```

\keys_set_filter:nnnN
\keys_set_filter:nnVN
\keys_set_filter:nnvN
\keys_set_filter:nnoN
\__keys_set_filter:nnnnN
\__keys_set_filter:onnN
\keys_set_filter:nnn
\keys_set_filter:nnV
\keys_set_filter:nnv
\keys_set_filter:nno
\__keys_set_filter:nnn
\keys_set_groups:nnn
\keys_set_groups:nnV
\keys_set_groups:nnv
\keys_set_groups:nno
\__keys_set_groups:nnn
\__keys_set_selective:nnn
\__keys_set_selective:nnnn
\__keys_set_selective:onnN
\__keys_set_selective:nn
11258 \cs_new_protected:Npn \keys_set_filter:nnnN
11259 { \__keys_set_filter:nnnnN \l__keys_unused_clist }
11260 \cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
11261 \cs_new_protected:Npn \__keys_set_filter:nnnnN #1#2#3#4#5
11262 {
11263   \clist_clear:N \l__keys_unused_clist
11264   \keys_set_filter:nnn {#2} {#3} {#4}
11265   \tl_set:Nx #5 { \exp_not:o { \l__keys_unused_clist } }
11266   \tl_set:Nn \l__keys_unused_clist {#1}
11267 }
11268 \cs_generate_variant:Nn \__keys_set_filter:nnnnN { o }
11269 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
11270 {
11271   \bool_if:NTF \l__keys_filtered_bool
11272   { \__keys_set_selective:nnn }
11273   { \__keys_set_filter:nnn }
11274   {#1} {#2} {#3}
11275 }
11276 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }

```

```

11277 \cs_new_protected:Npn \__keys_set_filter:nnn #1#2#3
11278 {
11279     \bool_set_true:N \l__keys_filtered_bool
11280     \__keys_set_selective:nnn {#1} {#2} {#3}
11281     \bool_set_false:N \l__keys_filtered_bool
11282 }
11283 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
11284 {
11285     \bool_if:NTF \l__keys_filtered_bool
11286     { \__keys_set_groups:nnn }
11287     { \__keys_set_selective:nnn }
11288     {#1} {#2} {#3}
11289 }
11290 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
11291 \cs_new_protected:Npn \__keys_set_groups:nnn #1#2#3
11292 {
11293     \bool_set_false:N \l__keys_filtered_bool
11294     \__keys_set_selective:nnn {#1} {#2} {#3}
11295     \bool_set_true:N \l__keys_filtered_bool
11296 }
11297 \cs_new_protected:Npn \__keys_set_selective:nnn
11298 { \__keys_set_selective:onnn \l__keys_selective_seq }
11299 \cs_new_protected:Npn \__keys_set_selective:nnnn #1#2#3#4
11300 {
11301     \seq_set_from_clist:Nn \l__keys_selective_seq {#3}
11302     \bool_if:NTF \l__keys_selective_bool
11303     { \keys_set:nn }
11304     { \__keys_set_selective:nn }
11305     {#2} {#4}
11306     \tl_set:Nn \l__keys_selective_seq {#1}
11307 }
11308 \cs_generate_variant:Nn \__keys_set_selective:nnnn { o }
11309 \cs_new_protected:Npn \__keys_set_selective:nn #1#2
11310 {
11311     \bool_set_true:N \l__keys_selective_bool
11312     \keys_set:nn {#1} {#2}
11313     \bool_set_false:N \l__keys_selective_bool
11314 }

```

(End definition for `\keys_set_filter:nnnN` and others. These functions are documented on page 175.)

<pre> __keys_set:n __keys_set:nn __keys_set_aux:nnn __keys_set_aux:onn __keys_find_key_module:w __keys_set_aux: __keys_set_selective: </pre>	<p>A shared system once again. First, set the current path and add a default if needed. There are then checks to see if the a value is required or forbidden. If everything passes, move on to execute the code.</p> <pre> 11315 \cs_new_protected:Npn __keys_set:n #1 11316 { 11317 \bool_set_true:N \l__keys_no_value_bool 11318 __keys_set_aux:onn \l__keys_module_tl {#1} { } 11319 } 11320 \cs_new_protected:Npn __keys_set:nn #1#2 11321 { 11322 \bool_set_false:N \l__keys_no_value_bool 11323 __keys_set_aux:onn \l__keys_module_tl {#1} {#2} 11324 } </pre>
---	--

The key path here can be fully defined, after which there is a search for the key and module names: the user may have passed them with part of what is actually the module (for our purposes) in the key name. As that happens on a per-key basis, we use the stack approach to restore the module name without a group.

```

11325 \cs_new_protected:Npn \__keys_set_aux:nnn #1#2#3
11326 {
11327   \tl_set:Nx \l_keys_path_tl
11328   {
11329     \tl_if_blank:nF {#1}
11330     { #1 / }
11331     \__keys_remove_spaces:n {#2}
11332   }
11333   \tl_clear:N \l__keys_module_tl
11334   \exp_after:wN \__keys_find_key_module:w \l_keys_path_tl / \q_stop
11335   \__keys_value_or_default:n {#3}
11336   \bool_if:NTF \l__keys_selective_bool
11337   { \__keys_set_selective: }
11338   { \__keys_execute: }
11339   \tl_set:Nn \l__keys_module_tl {#1}
11340 }
11341 \cs_generate_variant:Nn \__keys_set_aux:nnn { o }
11342 \cs_new_protected:Npn \__keys_find_key_module:w #1 / #2 \q_stop
11343 {
11344   \tl_if_blank:nTF {#2}
11345   { \tl_set:Nn \l_keys_key_tl {#1} }
11346   {
11347     \tl_put_right:Nx \l__keys_module_tl
11348     {
11349       \tl_if_empty:NF \l__keys_module_tl { / }
11350       #1
11351     }
11352     \__keys_find_key_module:w #2 \q_stop
11353   }
11354 }

```

If selective setting is active, there are a number of possible sub-cases to consider. The key name may not be known at all or if it is, it may not have any groups assigned. There is then the question of whether the selection is opt-in or opt-out.

```

11355 \cs_new_protected:Npn \__keys_set_selective:
11356 {
11357   \cs_if_exist:cTF { \c__keys_groups_root_tl \l_keys_path_tl }
11358   {
11359     \clist_set_eq:Nc \l__keys_groups_clist
11360     { \c__keys_groups_root_tl \l_keys_path_tl }
11361     \__keys_check_groups:
11362   }
11363   {
11364     \bool_if:NTF \l__keys_filtered_bool
11365     { \__keys_execute: }
11366     { \__keys_store_unused: }
11367   }
11368 }

```

In the case where selective setting requires a comparison of the list of groups which apply

to a key with the list of those which have been set active. That requires two mappings, and again a different outcome depending on whether opt-in or opt-out is set.

```

11369 \cs_new_protected:Npn \__keys_check_groups:
11370 {
11371   \bool_set_false:N \l__keys_tmp_bool
11372   \seq_map_inline:Nn \l__keys_selective_seq
11373   {
11374     \clist_map_inline:Nn \l__keys_groups_clist
11375     {
11376       \str_if_eq:nnT {##1} {####1}
11377       {
11378         \bool_set_true:N \l__keys_tmp_bool
11379         \clist_map_break:n { \seq_map_break: }
11380       }
11381     }
11382   }
11383   \bool_if:NTF \l__keys_tmp_bool
11384   {
11385     \bool_if:NTF \l__keys_filtered_bool
11386     { \__keys_store_unused: }
11387     { \__keys_execute: }
11388   }
11389   {
11390     \bool_if:NTF \l__keys_filtered_bool
11391     { \__keys_execute: }
11392     { \__keys_store_unused: }
11393   }
11394 }

```

(End definition for __keys_set:n and others.)

__keys_value_or_default:n If a value is given, return it as #1, otherwise send a default if available.

```

11395 \cs_new_protected:Npn \__keys_value_or_default:n #1
11396 {
11397   \bool_if:NTF \l__keys_no_value_bool
11398   {
11399     \cs_if_exist:cTF { \c__keys_default_root_tl \l_keys_path_tl }
11400     {
11401       \tl_set_eq:Nc
11402       \l_keys_value_tl
11403       { \c__keys_default_root_tl \l_keys_path_tl }
11404     }
11405     { \tl_clear:N \l_keys_value_tl }
11406   }
11407   { \tl_set:Nn \l_keys_value_tl {#1} }
11408 }

```

(End definition for __keys_value_or_default:n.)

__keys_execute: Actually executing a key is done in two parts. First, look for the key itself, then look for the **unknown** key with the same path. If both of these fail, complain. What exactly happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

```

11409 \cs_new_protected:Npn \__keys_execute:

```

```

11410 {
11411     \cs_if_exist:cTF { \c__keys_code_root_tl \l_keys_path_tl }
11412     {
11413         \cs_if_exist_use:c { \c__keys_validate_root_tl \l_keys_path_tl }
11414         \cs:w \c__keys_code_root_tl \l_keys_path_tl \exp_after:wN \cs_end:
11415         \exp_after:wN { \l_keys_value_tl }
11416     }
11417     { \__keys_execute_unknown: }
11418 }
11419 \cs_new_protected:Npn \__keys_execute_unknown:
11420 {
11421     \bool_if:NTF \l__keys_only_known_bool
11422     { \__keys_store_unused: }
11423     {
11424         \cs_if_exist:cTF
11425         { \c__keys_inherit_root_tl \__keys_parent:o \l_keys_path_tl }
11426         {
11427             \clist_map_inline:cn
11428             { \c__keys_inherit_root_tl \__keys_parent:o \l_keys_path_tl }
11429             {
11430                 \cs_if_exist:cT
11431                 { \c__keys_code_root_tl ##1 / \l_keys_key_tl }
11432                 {
11433                     \cs:w \c__keys_code_root_tl ##1 / \l_keys_key_tl
11434                     \exp_after:wN \cs_end: \exp_after:wN
11435                     { \l_keys_value_tl }
11436                     \clist_map_break:
11437                 }
11438             }
11439         }
11440         {
11441             \cs_if_exist:cTF
11442             { \c__keys_code_root_tl \l__keys_module_tl / unknown }
11443             {
11444                 \cs:w \c__keys_code_root_tl \l__keys_module_tl / unknown
11445                 \exp_after:wN \cs_end: \exp_after:wN { \l_keys_value_tl }
11446             }
11447             {
11448                 \__kernel_msg_error:nxxx { kernel } { key-unknown }
11449                 { \l_keys_path_tl } { \l__keys_module_tl }
11450             }
11451         }
11452     }
11453 }
11454 \cs_new:Npn \__keys_execute:nn #1#2
11455 {
11456     \cs_if_exist:cTF { \c__keys_code_root_tl #1 }
11457     {
11458         \cs:w \c__keys_code_root_tl #1 \exp_after:wN \cs_end:
11459         \exp_after:wN { \l_keys_value_tl }
11460     }
11461     {#2}
11462 }
11463 \cs_new_protected:Npn \__keys_store_unused:

```

```

11464 {
11465     \clist_put_right:Nx \l__keys_unused_clist
11466     {
11467         \exp_not:o \l_keys_key_tl
11468         \bool_if:NF \l__keys_no_value_bool
11469         { = { \exp_not:o \l_keys_value_tl } }
11470     }
11471 }

```

(End definition for __keys_execute: and others.)

__keys_choice_find:n Executing a choice has two parts. First, try the choice given, then if that fails call the
 __keys_multichoice_find:n unknown key. That always exists, as it is created when a choice is first made. So there
 is no need for any escape code. For multiple choices, the same code ends up used in a
 mapping.

```

11472 \cs_new:Npn \__keys_choice_find:n #1
11473 {
11474     \__keys_execute:nn { \l_keys_path_tl / \__keys_remove_spaces:n {#1} }
11475     { \__keys_execute:nn { \l_keys_path_tl / unknown } { } }
11476 }
11477 \cs_new:Npn \__keys_multichoice_find:n #1
11478 { \clist_map_function:nN {#1} \__keys_choice_find:n }

```

(End definition for __keys_choice_find:n and __keys_multichoice_find:n.)

20.7 Utilities

__keys_parent:n Used to strip off the ending part of the key path after the last /.

```

\__keys_parent:o
\__keys_parent:w
11479 \cs_new:Npn \__keys_parent:n #1
11480 { \__keys_parent:w #1 / / \q_stop { } }
11481 \cs_generate_variant:Nn \__keys_parent:n { o }
11482 \cs_new:Npn \__keys_parent:w #1 / #2 / #3 \q_stop #4
11483 {
11484     \tl_if_blank:nTF {#2}
11485     { \use_none:n #4 }
11486     {
11487         \__keys_parent:w #2 / #3 \q_stop { #4 / #1 }
11488     }
11489 }

```

(End definition for __keys_parent:n and __keys_parent:w.)

__keys_remove_spaces:n Used in a few places so worth handling as a dedicated function.

```

11490 \cs_new:Npn \__keys_remove_spaces:n #1
11491 { \tl_trim_spaces:o { \tl_to_str:n {#1} } }

```

(End definition for __keys_remove_spaces:n.)

\keys_if_exist_p:nn A utility for others to see if a key exists.

```

\keys_if_exist:nnTF
11492 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
11493 {
11494     \cs_if_exist:cTF
11495     { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 } }
11496     { \prg_return_true: }
11497     { \prg_return_false: }
11498 }

```

(End definition for `\keys_if_exist:nnTF`. This function is documented on page 175.)

`\keys_if_choice_exist_p:nnn` Just an alternative view on `\keys_if_exist:nnTF`.

```
\keys_if_choice_exist:nnnTF
11499 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
11500 { p , T , F , TF }
11501 {
11502   \cs_if_exist:cTF
11503     { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 / #3 } }
11504     { \prg_return_true: }
11505     { \prg_return_false: }
11506 }
```

(End definition for `\keys_if_choice_exist:nnnTF`. This function is documented on page 175.)

`\keys_show:nn` To show a key, show its code using a message.

`\keys_log:nn`

`__keys_show:Nnn`

```
11507 \cs_new_protected:Npn \keys_show:nn
11508 { \__keys_show:Nnn \msg_show:nnxxxx }
11509 \cs_new_protected:Npn \keys_log:nn
11510 { \__keys_show:Nnn \msg_log:nnxxxx }
11511 \cs_new_protected:Npn \__keys_show:Nnn #1#2#3
11512 {
11513   #1 { LaTeX / kernel } { show-key }
11514   { \__keys_remove_spaces:n { #2 / #3 } }
11515   {
11516     \keys_if_exist:nnT {#2} {#3}
11517     {
11518       \exp_args:Nnf \msg_show_item_unbraced:nn { code }
11519       {
11520         \exp_args:Nc \token_get_replacement_spec:N
11521         {
11522           \c__keys_code_root_tl
11523           \__keys_remove_spaces:n { #2 / #3 }
11524         }
11525       }
11526     }
11527   }
11528   { } { }
11529 }
```

(End definition for `\keys_show:nn`, `\keys_log:nn`, and `__keys_show:Nnn`. These functions are documented on page 175.)

20.8 Messages

For when there is a need to complain.

```
11530 \__kernel_msg_new:nnnn { kernel } { boolean-values-only }
11531 { Key~'#1'~accepts~boolean~values~only. }
11532 { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
11533 \__kernel_msg_new:nnnn { kernel } { key-choice-unknown }
11534 { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
11535 {
11536   The~key~'#1'~only~accepts~predefined~values,~
11537   and~'#2'~is~not~one~of~these.
11538 }
```

```

11539 \__kernel_msg_new:nnnn { kernel } { key-unknown }
11540 { The~key~'#1'~is~unknown~and~is~being~ignored. }
11541 {
11542   The~module~'#2'~does~not~have~a~key~called~'#1'.\\
11543   Check~that~you~have~spelled~the~key~name~correctly.
11544 }
11545 \__kernel_msg_new:nnnn { kernel } { nested-choice-key }
11546 { Attempt~to~define~'#1'~as~a~nested~choice~key. }
11547 {
11548   The~key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
11549   itself~a~choice.
11550 }
11551 \__kernel_msg_new:nnnn { kernel } { value-forbidden }
11552 { The~key~'#1'~does~not~take~a~value. }
11553 {
11554   The~key~'#1'~should~be~given~without~a~value.\\
11555   The~value~'#2'~was~present:~the~key~will~be~ignored.
11556 }
11557 \__kernel_msg_new:nnnn { kernel } { value-required }
11558 { The~key~'#1'~requires~a~value. }
11559 {
11560   The~key~'#1'~must~have~a~value.\\
11561   No~value~was~present:~the~key~will~be~ignored.
11562 }
11563 \__kernel_msg_new:nnn { kernel } { show-key }
11564 {
11565   The~key~#1~
11566   \tl_if_empty:nTF {#2}
11567     { is~undefined. }
11568     { has~the~properties: #2 . }
11569 }
11570 </initex | package>

```

21 l3fp implementation

Nothing to see here: everything is in the subfiles!

22 l3fp-aux implementation

```

11571 <(*initex | package)
11572 <@@=fp>

```

22.1 Access to primitives

__fp_int_eval:w Largely for performance reasons, we need to directly access primitives rather than use \int_eval:n. This happens *a lot*, so we use private names. The same is true for __fp_int_eval_end: \romannumeral, although it is used much less widely.

```

11573 \cs_new_eq:NN \__fp_int_eval:w \etex_numexpr:D
11574 \cs_new_eq:NN \__fp_int_eval_end: \scan_stop:
11575 \cs_new_eq:NN \__fp_int_to_roman:w \tex_romannumeral:D

```

(End definition for __fp_int_eval:w, __fp_int_eval_end:, and __fp_int_to_roman:w.)

22.2 Internal representation

Internally, a floating point number $\langle X \rangle$ is a token list containing

```
\s__fp \__fp_chk:w  $\langle case \rangle$   $\langle sign \rangle$   $\langle body \rangle$  ;
```

Let us explain each piece separately.

Internal floating point numbers are used in expressions, and in this context are subject to **f**-expansion. They must leave a recognizable mark after **f**-expansion, to prevent the floating point number from being re-parsed. Thus, `\s__fp` is simply another name for `\relax`.

When used directly without an accessor function, floating points should produce an error: this is the role of `__fp_chk:w`. We could make floating point variables be protected to prevent them from expanding under **x**-expansion, but it seems more convenient to treat them as a subcase of token list variables.

The (decimal part of the) IEEE-754-2008 standard requires the format to be able to represent special floating point numbers besides the usual positive and negative cases. We distinguish the various possibilities by their $\langle case \rangle$, which is a single digit:

- 0 zeros: `+0` and `-0`,
- 1 “normal” numbers (positive and negative),
- 2 infinities: `+inf` and `-inf`,
- 3 quiet and signalling `nan`.

The $\langle sign \rangle$ is 0 (positive) or 2 (negative), except in the case of `nan`, which have $\langle sign \rangle = 1$. This ensures that changing the $\langle sign \rangle$ digit to $2 - \langle sign \rangle$ is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

```
\s__fp \__fp_chk:w  $\langle case \rangle$   $\langle sign \rangle$  \s__fp... ;
```

where `\s__fp...` is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers ($\langle case \rangle = 1$) have the form

```
\s__fp \__fp_chk:w 1  $\langle sign \rangle$  { $\langle exponent \rangle$ } { $\langle X_1 \rangle$ } { $\langle X_2 \rangle$ } { $\langle X_3 \rangle$ } { $\langle X_4 \rangle$ } ;
```

Here, the $\langle exponent \rangle$ is an integer, between -10000 and 10000 . The body consists in four blocks of exactly 4 digits, $0000 \leq \langle X_i \rangle \leq 9999$, and the floating point is

$$(-1)^{\langle sign \rangle / 2} \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle \cdot 10^{\langle exponent \rangle - 16}$$

where we have concatenated the 16 digits. Currently, floating point numbers are normalized such that the $\langle exponent \rangle$ is minimal, in other words, $1000 \leq \langle X_1 \rangle \leq 9999$.

Calculations are done in base 10000, *i.e.* one myriad.

Table 1: Internal representation of floating point numbers.

Representation	Meaning
0 0 \s_fp_... ;	Positive zero.
0 2 \s_fp_... ;	Negative zero.
1 0 {\langle exponent\rangle} {\langle X ₁ \rangle} {\langle X ₂ \rangle} {\langle X ₃ \rangle} {\langle X ₄ \rangle} ;	Positive floating point.
1 2 {\langle exponent\rangle} {\langle X ₁ \rangle} {\langle X ₂ \rangle} {\langle X ₃ \rangle} {\langle X ₄ \rangle} ;	Negative floating point.
2 0 \s_fp_... ;	Positive infinity.
2 2 \s_fp_... ;	Negative infinity.
3 1 \s_fp_... ;	Quiet nan.
3 1 \s_fp_... ;	Signalling nan.

22.3 Using arguments and semicolons

_fp_use_none_stop_f:n This function removes an argument (typically a digit) and replaces it by \exp_stop_f:, a marker which stops f-type expansion.

```
11576 \cs_new:Npn \_fp\_use\_none\_stop\_f:n #1 { \exp\_stop\_f: }
```

(End definition for _fp_use_none_stop_f:n.)

_fp_use_s:n Those functions place a semicolon after one or two arguments (typically digits).

```
\_fp\_use\_s:nn
11577 \cs_new:Npn \_fp\_use\_s:n #1 { #1; }
11578 \cs_new:Npn \_fp\_use\_s:nn #1#2 { #1#2; }
```

(End definition for _fp_use_s:n and _fp_use_s:nn.)

_fp_use_none_until_s:w Those functions select specific arguments among a set of arguments delimited by a semicolon.

```
\_fp\_use\_i\_until\_s:nw
\_fp\_use\_ii\_until\_s:nnw
11579 \cs_new:Npn \_fp\_use\_none\_until\_s:w #1; { }
11580 \cs_new:Npn \_fp\_use\_i\_until\_s:nw #1#2; {#1}
11581 \cs_new:Npn \_fp\_use\_ii\_until\_s:nnw #1#2#3; {#2}
```

(End definition for _fp_use_none_until_s:w, _fp_use_i_until_s:nw, and _fp_use_ii_until_s:nnw.)

_fp_reverse_args:Nww Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.

```
11582 \cs_new:Npn \_fp\_reverse\_args:Nww #1 #2; #3; { #1 #3; #2; }
```

(End definition for _fp_reverse_args:Nww.)

_fp_rrot:www Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive ROT, hence the name.

```
11583 \cs_new:Npn \_fp\_rrot:www #1; #2; #3; { #2; #3; #1; }
```

(End definition for _fp_rrot:www.)

_fp_use_i:ww Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.

```
\_fp\_use\_i:www
11584 \cs_new:Npn \_fp\_use\_i:ww #1; #2; { #1; }
11585 \cs_new:Npn \_fp\_use\_i:www #1; #2; #3; { #1; }
```

(End definition for _fp_use_i:ww and _fp_use_i:www.)

22.4 Constants, and structure of floating points

`__fp_misused:n` This receives a floating point object (floating point number or tuple) and generates an error stating that it was misused. This is called when for instance an `fp` variable is left in the input stream and its contents reach T_EX's stomach.

```
11586 \cs_new_protected:Npn \__fp_misused:n #1
11587 { \__kernel_msg_error:nnx { kernel } { misused-fp } { \fp_to_tl:n {#1} } }
```

(End definition for `__fp_misused:n`.)

`\s__fp` Floating points numbers all start with `\s__fp __fp_chk:w`, where `\s__fp` is equal to the T_EX primitive `\relax`, and `__fp_chk:w` is protected. The rest of the floating point number is made of characters (or `\relax`). This ensures that nothing expands under f-expansion, nor under x-expansion. However, when typeset, `\s__fp` does nothing, and `__fp_chk:w` is expanded. We define `__fp_chk:w` to produce an error.

```
11588 \scan_new:N \s__fp
11589 \cs_new_protected:Npn \__fp_chk:w #1 ;
11590 { \__fp_misused:n { \s__fp \__fp_chk:w #1 ; } }
```

(End definition for `\s__fp` and `__fp_chk:w`.)

`\s__fp_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

```
\s__fp_stop
11591 \scan_new:N \s__fp_mark
11592 \scan_new:N \s__fp_stop
```

(End definition for `\s__fp_mark` and `\s__fp_stop`.)

`\s__fp_invalid` A couple of scan marks used to indicate where special floating point numbers come from.

```
\s__fp_underflow
\s__fp_overflow
\s__fp_division
\s__fp_exact
11593 \scan_new:N \s__fp_invalid
11594 \scan_new:N \s__fp_underflow
11595 \scan_new:N \s__fp_overflow
11596 \scan_new:N \s__fp_division
11597 \scan_new:N \s__fp_exact
```

(End definition for `\s__fp_invalid` and others.)

`\c_zero_fp` The special floating points. We define the floating points here as “exact”.
`\c_minus_zero_fp`
`\c_inf_fp`
`\c_minus_inf_fp`
`\c_nan_fp`

```
11598 \tl_const:Nn \c_zero_fp { \s__fp \__fp_chk:w 0 0 \s__fp_exact ; }
11599 \tl_const:Nn \c_minus_zero_fp { \s__fp \__fp_chk:w 0 2 \s__fp_exact ; }
11600 \tl_const:Nn \c_inf_fp { \s__fp \__fp_chk:w 2 0 \s__fp_exact ; }
11601 \tl_const:Nn \c_minus_inf_fp { \s__fp \__fp_chk:w 2 2 \s__fp_exact ; }
11602 \tl_const:Nn \c_nan_fp { \s__fp \__fp_chk:w 3 1 \s__fp_exact ; }
```

(End definition for `\c_zero_fp` and others. These variables are documented on page 185.)

`\c__fp_prec_int` The number of digits of floating points.
`\c__fp_half_prec_int`
`\c__fp_block_int`

```
11603 \int_const:Nn \c__fp_prec_int { 16 }
11604 \int_const:Nn \c__fp_half_prec_int { 8 }
11605 \int_const:Nn \c__fp_block_int { 4 }
```

(End definition for `\c__fp_prec_int`, `\c__fp_half_prec_int`, and `\c__fp_block_int`.)

`\c__fp_myriad_int` Blocks have 4 digits so this integer is useful.

```
11606 \int_const:Nn \c__fp_myriad_int { 10000 }
```


(End definition for `\c__fp_myriad_int`.)

`\c__fp_minus_min_exponent_int` `\c__fp_max_exponent_int` Normal floating point numbers have an exponent between `-minus_min_exponent` and `max_exponent` inclusive. Larger numbers are rounded to $\pm\infty$. Smaller numbers are rounded to ± 0 . It would be more natural to define a `min_exponent` with the opposite sign but that would waste one T_EX count.

```
11607 \int_const:Nn \c__fp_minus_min_exponent_int { 10000 }
11608 \int_const:Nn \c__fp_max_exponent_int { 10000 }
```

(End definition for `\c__fp_minus_min_exponent_int` and `\c__fp_max_exponent_int`.)

`\c__fp_max_exp_exponent_int` If a number's exponent is larger than that, its exponential overflows/underflows.

```
11609 \int_const:Nn \c__fp_max_exp_exponent_int { 5 }
```

(End definition for `\c__fp_max_exp_exponent_int`.)

`\c__fp_overflowing_fp` A floating point number that is bigger than all normal floating point numbers. This replaces infinities when converting to formats that do not support infinities.

```
11610 \tl_const:Nx \c__fp_overflowing_fp
11611 {
11612   \s__fp \__fp_chk:w 1 0
11613   { \int_eval:n { \c__fp_max_exponent_int + 1 } }
11614   {1000} {0000} {0000} {0000} ;
11615 }
```

(End definition for `\c__fp_overflowing_fp`.)

`__fp_zero_fp:N` `__fp_inf_fp:N` In case of overflow or underflow, we have to output a zero or infinity with a given sign.

```
11616 \cs_new:Npn \__fp_zero_fp:N #1
11617 { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
11618 \cs_new:Npn \__fp_inf_fp:N #1
11619 { \s__fp \__fp_chk:w 2 #1 \s__fp_overflow ; }
```

(End definition for `__fp_zero_fp:N` and `__fp_inf_fp:N`.)

`__fp_exponent:w` For normal numbers, the function expands to the exponent, otherwise to 0. This is used in l3str-format.

```
11620 \cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
11621 {
11622   \if_meaning:w 1 #1
11623     \exp_after:wN \__fp_use_ii_until_s:nnw
11624   \else:
11625     \exp_after:wN \__fp_use_i_until_s:nw
11626     \exp_after:wN 0
11627   \fi:
11628 }
```

(End definition for `__fp_exponent:w`.)

`__fp_neg_sign:N` When appearing in an integer expression or after `\int_value:w`, this expands to the sign opposite to #1, namely 0 (positive) is turned to 2 (negative), 1 (nan) to 1, and 2 to 0.

```
11629 \cs_new:Npn \__fp_neg_sign:N #1
11630 { \__fp_int_eval:w 2 - #1 \__fp_int_eval_end: }
```

(End definition for `__fp_neg_sign:N`.)

22.5 Overflow, underflow, and exact zero

`__fp_sanitizew` expects the sign and the exponent in some order, then the significand (which we don't touch). Outputs the corresponding floating point number, possibly underflowed to ± 0 or overflowed to $\pm\infty$. The functions `__fp_underfloww` and `__fp_overfloww` are defined in `l3fp-traps`.

```

11631 \cs_new:Npn \__fp_sanitizew #1 #2;
11632 {
11633   \if_case:w
11634     \if_int_compare:w #2 > \c__fp_max_exponent_int 1 ~ \else:
11635     \if_int_compare:w #2 < - \c__fp_minus_min_exponent_int 2 ~ \else:
11636     \if_meaning:w 1 #1 3 ~ \fi: \fi: \fi: 0 ~
11637     \or: \exp_after:wN \__fp_overflow:w
11638     \or: \exp_after:wN \__fp_underflow:w
11639     \or: \exp_after:wN \__fp_sanitizew
11640     \fi:
11641     \s__fp \__fp_chk:w 1 #1 {#2}
11642   }
11643 \cs_new:Npn \__fp_sanitizewN #1; #2 { \__fp_sanitizew #2 #1; }
11644 \cs_new:Npn \__fp_sanitizew_zero:w \s__fp \__fp_chk:w #1 #2 #3;
11645   { \c_zero_fp }

```

(End definition for `__fp_sanitizew`, `__fp_sanitizewN`, and `__fp_sanitizew_zero:w`.)

22.6 Expanding after a floating point number

`__fp_exp_after_ow`
`__fp_exp_after_fnw`

`__fp_exp_after_ow` *<floating point>*
`__fp_exp_after_fnw` *{<tokens>}* *<floating point>*

Places *<tokens>* (empty in the case of `__fp_exp_after_ow`) between the *<floating point>* and the following tokens, then hits those tokens with `o` or `f`-expansion, and leaves the floating point number unchanged.

We first distinguish normal floating points, which have a significand, from the much simpler special floating points.

```

11646 \cs_new:Npn \__fp_exp_after_ow \s__fp \__fp_chk:w #1
11647 {
11648   \if_meaning:w 1 #1
11649     \exp_after:wN \__fp_exp_after_normal:nNNw
11650   \else:
11651     \exp_after:wN \__fp_exp_after_special:nNNw
11652   \fi:
11653   { }
11654   #1
11655 }
11656 \cs_new:Npn \__fp_exp_after_fnw #1 \s__fp \__fp_chk:w #2
11657 {
11658   \if_meaning:w 1 #2
11659     \exp_after:wN \__fp_exp_after_normal:nNNw
11660   \else:
11661     \exp_after:wN \__fp_exp_after_special:nNNw
11662   \fi:
11663   { \exp:w \exp_end_continue_f:w #1 }
11664   #2
11665 }

```

(End definition for `_fp_exp_after_o:w` and `_fp_exp_after_f:nw`.)

```
\_fp_exp_after_special:nNNw      \_fp_exp_after_special:nNNw {<after>} <case> <sign> <scan mark> ;
Special floating point numbers are easy to jump over since they contain few tokens.
11666 \cs_new:Npn \_fp_exp_after_special:nNNw #1#2#3#4;
11667   {
11668     \exp_after:wN \_s\_fp
11669     \exp_after:wN \_fp_chk:w
11670     \exp_after:wN #2
11671     \exp_after:wN #3
11672     \exp_after:wN #4
11673     \exp_after:wN ;
11674     #1
11675   }
```

(End definition for `_fp_exp_after_special:nNNw`.)

`_fp_exp_after_normal:nNNw` For normal floating point numbers, life is slightly harder, since we have many tokens to jump over. Here it would be slightly better if the digits were not braced but instead were delimited arguments (for instance delimited by `,`). That may be changed some day.

```
11676 \cs_new:Npn \_fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;
11677   {
11678     \exp_after:wN \_fp_exp_after_normal:Nwwwww
11679     \exp_after:wN #2
11680     \int_value:w #3 \exp_after:wN ;
11681     \int_value:w 1 #4 \exp_after:wN ;
11682     \int_value:w 1 #5 \exp_after:wN ;
11683     \int_value:w 1 #6 \exp_after:wN ;
11684     \int_value:w 1 #7 \exp_after:wN ; #1
11685   }
11686 \cs_new:Npn \_fp_exp_after_normal:Nwwwww
11687   #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
11688   { \_s\_fp \_fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }
```

(End definition for `_fp_exp_after_normal:nNNw`.)

22.7 Other floating point types

`\s_fp_tuple` Floating point tuples take the form `\s_fp_tuple _fp_tuple_chk:w { <fp 1> <fp 2> ... }` ; where each `<fp>` is a floating point number or tuple, hence ends with `;` itself. When a tuple is typeset, `_fp_tuple_chk:w` produces an error, just like usual floating point numbers. Tuples may have zero or one element.

```
11689 \scan_new:N \s\_fp_tuple
11690 \cs_new_protected:Npn \_fp_tuple_chk:w #1 ;
11691   { \_fp_misused:n { \s\_fp_tuple \_fp_tuple_chk:w #1 ; } }
11692 \tl_const:Nn \c\_fp_empty_tuple_fp
11693   { \s\_fp_tuple \_fp_tuple_chk:w { } ; }
```

(End definition for `\s_fp_tuple`, `_fp_tuple_chk:w`, and `\c_fp_empty_tuple_fp`.)

`_fp_tuple_count:w` Count the number of items in a tuple of floating points by counting semicolons. The
`_fp_array_count:n` technique is very similar to `\tl_count:n`, but with the loop built-in. Checking for the
`_fp_tuple_count_loop:Nw` end of the loop is done with the `\use_none:n #1` construction.

```

11694 \cs_new:Npn \__fp_array_count:n #1
11695 { \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w {#1} ; }
11696 \cs_new:Npn \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w #1 ;
11697 {
11698   \int_value:w \__fp_int_eval:w 0
11699   \__fp_tuple_count_loop:Nw #1 { ? \prg_break: } ;
11700   \prg_break_point:
11701   \__fp_int_eval_end:
11702 }
11703 \cs_new:Npn \__fp_tuple_count_loop:Nw #1#2;
11704 { \use_none:n #1 + 1 \__fp_tuple_count_loop:Nw }

```

(End definition for __fp_tuple_count:w, __fp_array_count:n, and __fp_tuple_count_loop:Nw.)

__fp_if_type_fp:NTwFw Used as __fp_if_type_fp:NTwFw <marker> {<true code>} \s__fp {<false code>} \q_stop, this test whether the <marker> is \s__fp or not and runs the appropriate <code>. The very unusual syntax is for optimization purposes as that function is used for all floating point operations.

```

11705 \cs_new:Npn \__fp_if_type_fp:NTwFw #1 \s__fp #2 #3 \q_stop {#2}

```

(End definition for __fp_if_type_fp:NTwFw.)

__fp_array_if_all_fp:nTF True if all items are floating point numbers. Used for min.
__fp_array_if_all_fp_loop:w

```

11706 \cs_new:Npn \__fp_array_if_all_fp:nTF #1
11707 {
11708   \__fp_array_if_all_fp_loop:w #1 { \s__fp \prg_break: } ;
11709   \prg_break_point: \use_i:nn
11710 }
11711 \cs_new:Npn \__fp_array_if_all_fp_loop:w #1#2 ;
11712 {
11713   \__fp_if_type_fp:NTwFw
11714   #1 \__fp_array_if_all_fp_loop:w
11715   \s__fp { \prg_break:n \use_iii:nnn }
11716   \q_stop
11717 }

```

(End definition for __fp_array_if_all_fp:nTF and __fp_array_if_all_fp_loop:w.)

__fp_type_from_scan:N Used as __fp_type_from_scan:N <token>. Grabs the pieces of the stringified <token> which lies after the first s__fp. If the <token> does not contain that string, the result is _?.
__fp_type_from_scan_other:N
__fp_type_from_scan:w

```

11718 \cs_new:Npn \__fp_type_from_scan:N #1
11719 {
11720   \__fp_if_type_fp:NTwFw
11721   #1 { }
11722   \s__fp { \__fp_type_from_scan_other:N #1 }
11723   \q_stop
11724 }
11725 \cs_new:Npx \__fp_type_from_scan_other:N #1
11726 {
11727   \exp_not:N \exp_after:wN \exp_not:N \__fp_type_from_scan:w
11728   \exp_not:N \token_to_str:N #1 \exp_not:N \q_mark
11729   \tl_to_str:n { s__fp _? } \exp_not:N \q_mark \exp_not:N \q_stop
11730 }

```

```

11731 \exp_last_unbraced:NNNNo
11732 \cs_new:Npn \__fp_type_from_scan:w #1
11733 { \tl_to_str:n { s__fp } } #2 \q_mark #3 \q_stop {#2}

```

(End definition for __fp_type_from_scan:N, __fp_type_from_scan_other:N, and __fp_type_from_scan:w.)

```

\__fp_change_func_type:NNN
\__fp_change_func_type_aux:w
\__fp_change_func_type_chk:NNN

```

Arguments are $\langle type\ marker \rangle$ $\langle function \rangle$ $\langle recovery \rangle$. This gives the function obtained by placing the type after @@. If the function is not defined then $\langle recovery \rangle$ $\langle function \rangle$ is used instead; however that test is not run when the $\langle type\ marker \rangle$ is $\backslash s_fp$.

```

11734 \cs_new:Npn \__fp_change_func_type:NNN #1#2#3
11735 {
11736   \__fp_if_type_fp:NTwFw
11737   #1 #2
11738   \s__fp
11739   {
11740     \exp_after:wN \__fp_change_func_type_chk:NNN
11741     \cs:w
11742     __fp \__fp_type_from_scan_other:N #1
11743     \exp_after:wN \__fp_change_func_type_aux:w \token_to_str:N #2
11744     \cs_end:
11745     #2 #3
11746   }
11747   \q_stop
11748 }
11749 \exp_last_unbraced:NNNNo
11750 \cs_new:Npn \__fp_change_func_type_aux:w #1 { \tl_to_str:n { __fp } } { }
11751 \cs_new:Npn \__fp_change_func_type_chk:NNN #1#2#3
11752 {
11753   \if_meaning:w \scan_stop: #1
11754   \exp_after:wN #3 \exp_after:wN #2
11755   \else:
11756   \exp_after:wN #1
11757   \fi:
11758 }

```

(End definition for __fp_change_func_type:NNN, __fp_change_func_type_aux:w, and __fp_change_func_type_chk:NNN.)

```

\__fp_exp_after_any_f:Nnw
\__fp_exp_after_any_f:nw
\__fp_exp_after_stop_f:nw

```

The Nnw function simply dispatches to the appropriate $\backslash _fp_exp_after \dots_f:nw$ with “...” (either empty or $_ \langle type \rangle$) extracted from #1, which should start with $\backslash s_fp$. If it doesn’t start with $\backslash s_fp$ the function $\backslash _fp_exp_after_?_f:nw$ defined in `l3fp-parse` gives an error; another special $\langle type \rangle$ is `stop`, useful for loops, see below. The nw function has an important optimization for floating points numbers; it also fetches its type marker #2 from the floating point.

```

11759 \cs_new:Npn \__fp_exp_after_any_f:Nnw #1
11760 { \cs:w __fp_exp_after \__fp_type_from_scan_other:N #1 _f:nw \cs_end: }
11761 \cs_new:Npn \__fp_exp_after_any_f:nw #1#2
11762 {
11763   \__fp_if_type_fp:NTwFw
11764   #2 \__fp_exp_after_f:nw
11765   \s__fp { \__fp_exp_after_any_f:Nnw #2 }
11766   \q_stop
11767   {#1} #2

```

```

11768 }
11769 \cs_new_eq:NN \__fp_exp_after_stop_f:nw \use_none:nn

```

(End definition for `__fp_exp_after_any_f:Nnw`, `__fp_exp_after_any_f:nw`, and `__fp_exp_after_stop_f:nw`.)

```

\__fp_exp_after_tuple_o:w
\__fp_exp_after_tuple_f:nw
\__fp_exp_after_array_f:w

```

The loop works by using the `n` argument of `__fp_exp_after_any_f:nw` to place the loop macro after the next item in the tuple and expand it.

```

\__fp_exp_after_array_f:w
⟨fp1⟩ ;
...
⟨fpn⟩ ;
\s__fp_stop

```

```

11770 \cs_new:Npn \__fp_exp_after_tuple_o:w
11771 { \__fp_exp_after_tuple_f:nw { \exp_after:wN \exp_stop_f: } }
11772 \cs_new:Npn \__fp_exp_after_tuple_f:nw
11773 #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
11774 {
11775   \exp_after:wN \s__fp_tuple
11776   \exp_after:wN \__fp_tuple_chk:w
11777   \exp_after:wN {
11778     \exp:w \exp_end_continue_f:w
11779     \__fp_exp_after_array_f:w #2 \s__fp_stop
11780   \exp_after:wN }
11781   \exp_after:wN ;
11782   \exp:w \exp_end_continue_f:w #1
11783 }
11784 \cs_new:Npn \__fp_exp_after_array_f:w
11785 { \__fp_exp_after_any_f:nw { \__fp_exp_after_array_f:w } }

```

(End definition for `__fp_exp_after_tuple_o:w`, `__fp_exp_after_tuple_f:nw`, and `__fp_exp_after_array_f:w`.)

22.8 Packing digits

When a positive integer `#1` is known to be less than 10^8 , the following trick splits it into two blocks of 4 digits, padding with zeros on the left.

```

\cs_new:Npn \pack:NNNNnw #1 #2#3#4#5 #6; { {#2#3#4#5} {#6} }
\exp_after:wN \pack:NNNNnw
\__fp_int_value:w \__fp_int_eval:w 1 0000 0000 + #1 ;

```

The idea is that adding 10^8 to the number ensures that it has exactly 9 digits, and can then easily find which digits correspond to what position in the number. Of course, this can be modified for any number of digits less or equal to 9 (we are limited by $\text{T}_{\text{E}}\text{X}$'s integers). This method is very heavily relied upon in `l3fp-basics`.

More specifically, the auxiliary inserts `+ #1#2#3#4#5 ; {#6}`, which allows us to compute several blocks of 4 digits in a nested manner, performing carries on the fly. Say we want to compute 12345×66778899 . With simplified names, we would do

```

\exp_after:wN \post_processing:w
\__fp_int_value:w \__fp_int_eval:w - 5 0000
\exp_after:wN \pack:NNNNNw
\__fp_int_value:w \__fp_int_eval:w 4 9995 0000
+ 12345 * 6677
\exp_after:wN \pack:NNNNNw
\__fp_int_value:w \__fp_int_eval:w 5 0000 0000
+ 12345 * 8899 ;

```

The `\exp_after:wN` triggers `\int_value:w __fp_int_eval:w`, which starts a first computation, whose initial value is $-5\,0000$ (the “leading shift”). In that computation appears an `\exp_after:wN`, which triggers the nested computation `\int_value:w __fp_int_eval:w` with starting value $4\,9995\,0000$ (the “middle shift”). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation’s value is $5\,0000\,0000 + 12345 \times 8899$, which has 9 digits. Adding $5 \cdot 10^8$ to the product allowed us to know how many digits to expect as long as the numbers to multiply are not too big; it also works to some extent with negative results. The `pack` function puts the last 4 of those 9 digits into a brace group, moves the semi-colon delimiter, and inserts a `+`, which combines the carry with the previous computation. The shifts nicely combine into $5\,0000\,0000/10^4 + 4\,9995\,0000 = 5\,0000\,0000$. As long as the operands are in some range, the result of this second computation has 9 digits. The corresponding `pack` function, expanded after the result is computed, braces the last 4 digits, and leaves `+ <5 digits>` for the initial computation. The “leading shift” cancels the combination of the other shifts, and the `\post_processing:w` takes care of packing the last few digits.

Admittedly, this is quite intricate. It is probably the key in making `l3fp` as fast as other pure \TeX floating point units despite its increased precision. In fact, this is used so much that we provide different sets of packing functions and shifts, depending on ranges of input.

```

\__fp_pack:NNNNNw
\c__fp_trailing_shift_int
\c__fp_middle_shift_int
\c__fp_leading_shift_int
11786 \int_const:Nn \c__fp_leading_shift_int { - 5 0000 }
11787 \int_const:Nn \c__fp_middle_shift_int { 5 0000 * 9999 }
11788 \int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
11789 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }

```

This set of shifts allows for computations involving results in the range $[-4 \cdot 10^8, 5 \cdot 10^8 - 1]$. Shifted values all have exactly 9 digits.

(End definition for `__fp_pack:NNNNNw` and others.)

```

\__fp_pack_big:NNNNNNw
\c__fp_big_trailing_shift_int
\c__fp_big_middle_shift_int
\c__fp_big_leading_shift_int

```

This set of shifts allows for computations involving results in the range $[-5 \cdot 10^8, 6 \cdot 10^8 - 1]$ (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper bound is due to \TeX ’s limit of $2^{31} - 1$ on integers. The shifts are chosen to be roughly the mid-point of 10^9 and 2^{31} , the two bounds on 10-digit integers in \TeX .

```

11790 \int_const:Nn \c__fp_big_leading_shift_int { - 15 2374 }
11791 \int_const:Nn \c__fp_big_middle_shift_int { 15 2374 * 9999 }
11792 \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
11793 \cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
11794 { + #1#2#3#4#5#6 ; {#7} }

```

(End definition for `__fp_pack_big:NNNNNNw` and others.)

```

\__fp_pack_Bigg:NNNNNNw
\__fp_Bigg_trailing_shift_int
\__fp_Bigg_middle_shift_int
\__fp_Bigg_leading_shift_int
11795 \int_const:Nn \__fp_Bigg_leading_shift_int { - 20 0000 }
11796 \int_const:Nn \__fp_Bigg_middle_shift_int { 20 0000 * 9999 }
11797 \int_const:Nn \__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
11798 \cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
11799 { + #1#2#3#4#5#6 ; {#7} }

```

This set of shifts allows for computations with results in the range $[-1 \cdot 10^9, 147483647]$; the end-point is $2^{31} - 1 - 2 \cdot 10^9 \simeq 1.47 \cdot 10^8$. Shifted values all have exactly 10 digits.

(End definition for `__fp_pack_Bigg:NNNNNNw` and others.)

```

\__fp_pack_twice_four:wNNNNNNNN
\__fp_pack_twice_four:wNNNNNNNN <tokens> ; <≥ 8 digits>
Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting
several copies of this function before a semicolon packs more digits since each takes the
digits packed by the others in its first argument.
11800 \cs_new:Npn \__fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
11801 { #1 {#2#3#4#5} {#6#7#8#9} ; }

```

(End definition for `__fp_pack_twice_four:wNNNNNNNN`.)

```

\__fp_pack_eight:wNNNNNNNN
\__fp_pack_eight:wNNNNNNNN <tokens> ; <≥ 8 digits>
Grabs one set of 8 digits and places them before the semi-colon delimiter as a single
group. Putting several copies of this function before a semicolon packs more digits since
each takes the digits packed by the others in its first argument.
11802 \cs_new:Npn \__fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
11803 { #1 {#2#3#4#5#6#7#8#9} ; }

```

(End definition for `__fp_pack_eight:wNNNNNNNN`.)

```

\__fp_basics_pack_low:NNNNNw
\__fp_basics_pack_high:NNNNNw
\__fp_basics_pack_high_carry:w
Addition and multiplication of significands are done in two steps: first compute a (more or
less) exact result, then round and pack digits in the final (braced) form. These functions
take care of the packing, with special attention given to the case where rounding has
caused a carry. Since rounding can only shift the final digit by 1, a carry always produces
an exact power of 10. Thus, \__fp_basics_pack_high_carry:w is always followed by
four times {0000}.

```

This is used in `l3fp-basics` and `l3fp-extended`.

```

11804 \cs_new:Npn \__fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
11805 { + #1 - 1 ; {#2#3#4#5} {#6} ; }
11806 \cs_new:Npn \__fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
11807 {
11808   \if_meaning:w 2 #1
11809     \__fp_basics_pack_high_carry:w
11810   \fi:
11811   ; {#2#3#4#5} {#6}
11812 }
11813 \cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
11814 { \fi: + 1 ; {1000} }

```

(End definition for `__fp_basics_pack_low:NNNNNw`, `__fp_basics_pack_high:NNNNNw`, and `__fp_basics_pack_high_carry:w`.)

`_fp_basics_pack_weird_low:NNNNw`
`_fp_basics_pack_weird_high:NNNNNNNNw`

This is used in l3fp-basics for additions and divisions. Their syntax is confusing, hence the name.

```

11815 \cs_new:Npn \_fp\_basics\_pack\_weird\_low:NNNNw #1 #2#3#4 #5;
11816 {
11817   \if_meaning:w 2 #1
11818     + 1
11819   \fi:
11820   \_fp\_int\_eval\_end:
11821   #2#3#4; {#5} ;
11822 }
11823 \cs_new:Npn \_fp\_basics\_pack\_weird\_high:NNNNNNNNw
11824   1 #1#2#3#4 #5#6#7#8 #9; { ; {#1#2#3#4} {#5#6#7#8} {#9} }

```

(End definition for `_fp_basics_pack_weird_low:NNNNw` and `_fp_basics_pack_weird_high:NNNNNNNNw`.)

22.9 Decimate (dividing by a power of 10)

`_fp_decimate:nNnnnn`

`_fp_decimate:nNnnnn {<shift>} {<f1>}`
`{<X1>} {<X2>} {<X3>} {<X4>}`

Each $\langle X_i \rangle$ consists in 4 digits exactly, and $1000 \leq \langle X_1 \rangle < 9999$. The first argument determines by how much we shift the digits. $\langle f_1 \rangle$ is called as follows:

$\langle f_1 \rangle$ $\langle \text{rounding} \rangle$ $\{ \langle X'_1 \rangle \}$ $\{ \langle X'_2 \rangle \}$ $\langle \text{extra-digits} \rangle$;

where $0 \leq \langle X'_i \rangle < 10^8 - 1$ are 8 digit integers, forming the truncation of our number. In other words,

$$\left(\sum_{i=1}^4 \langle X_i \rangle \cdot 10^{-4i} \cdot 10^{-\langle \text{shift} \rangle} \right) - (\langle X'_1 \rangle \cdot 10^{-8} + \langle X'_2 \rangle \cdot 10^{-16}) = 0. \langle \text{extra-digits} \rangle \cdot 10^{-16} \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The $\langle \text{rounding} \rangle$ digit is 0 if and only if the difference is exactly 0, and 5 if and only if the difference is exactly $0.5 \cdot 10^{-16}$. Otherwise, it is the (non-0, non-5) digit closest to 10^{17} times the difference. In particular, if the shift is 17 or more, all the digits are dropped, $\langle \text{rounding} \rangle$ is 1 (not 0), and $\langle X'_1 \rangle$ and $\langle X'_2 \rangle$ are both zero.

If the shift is 1, the $\langle \text{rounding} \rangle$ digit is simply the only digit that was pushed out of the brace groups (this is important for subtraction). It would be more natural for the $\langle \text{rounding} \rangle$ digit to be placed after the $\langle X'_i \rangle$, but the choice we make involves less reshuffling.

Note that this function treats negative $\langle \text{shift} \rangle$ as 0.

```

11825 \cs_new:Npn \_fp\_decimate:nNnnnn #1
11826 {
11827   \cs:w
11828     \_fp\_decimate\_
11829     \if_int_compare:w \_fp\_int\_eval:w #1 > \c\_fp\_prec\_int
11830       tiny
11831     \else:
11832       \_fp\_int\_to\_roman:w \_fp\_int\_eval:w #1
11833     \fi:
11834     :Nnnnn
11835   \cs_end:
11836 }

```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.

(End definition for `_fp_decimate:nNnnnn`.)

If the $\langle shift \rangle$ is zero, or too big, life is very easy.

```
\_fp\_decimate_:Nnnnn
\_fp\_decimate\_tiny:Nnnnn
11837 \cs_new:Npn \_fp\_decimate_:Nnnnn #1 #2#3#4#5
11838 { #1 0 {#2#3} {#4#5} ; }
11839 \cs_new:Npn \_fp\_decimate\_tiny:Nnnnn #1 #2#3#4#5
11840 { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 ; }
```

(End definition for `_fp_decimate_:Nnnnn` and `_fp_decimate_tiny:Nnnnn`.)

```
\_fp\_decimate\_auxi:Nnnnn      \_fp\_decimate\_auxi:Nnnnn  $\langle f_1 \rangle$  { $\langle X_1 \rangle$ } { $\langle X_2 \rangle$ } { $\langle X_3 \rangle$ } { $\langle X_4 \rangle$ }
\_fp\_decimate\_auxii:Nnnnn      Shifting happens in two steps: compute the  $\langle rounding \rangle$  digit, and repack digits into
\_fp\_decimate\_auxiii:Nnnnn      two blocks of 8. The sixteen functions are very similar, and defined through \_fp\_
\_fp\_decimate\_auxiv:Nnnnn      tmp:w. The arguments are as follows: #1 indicates which function is being defined; after
\_fp\_decimate\_auxv:Nnnnn      one step of expansion, #2 yields the “extra digits” which are then converted by \_fp\_
\_fp\_decimate\_auxvi:Nnnnn      round\_digit:Nw to the  $\langle rounding \rangle$  digit (note the + separating blocks of digits to
\_fp\_decimate\_auxvii:Nnnnn      avoid overflowing TeX’s integers). This triggers the f-expansion of \_fp\_decimate\_
\_fp\_decimate\_auxviii:Nnnnn     pack:nnnnnnnnnw,8 responsible for building two blocks of 8 digits, and removing the
\_fp\_decimate\_auxix:Nnnnn      rest. For this to work, #3 alternates between braced and unbraced blocks of 4 digits, in
\_fp\_decimate\_auxx:Nnnnn      such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.
\_fp\_decimate\_auxxi:Nnnnn
\_fp\_decimate\_auxxii:Nnnnn
\_fp\_decimate\_auxxiii:Nnnnn
\_fp\_decimate\_auxxiv:Nnnnn
\_fp\_decimate\_auxxv:Nnnnn
\_fp\_decimate\_auxxvi:Nnnnn
11841 \cs_new:Npn \_fp\_tmp:w #1 #2 #3
11842 {
11843   \cs_new:cpn { \_fp\_decimate\_ #1 :Nnnnn } ##1 ##2##3##4##5
11844   {
11845     \exp_after:wN ##1
11846     \int_value:w
11847     \exp_after:wN \_fp\_round\_digit:Nw #2 ;
11848     \_fp\_decimate\_pack:nnnnnnnnnw #3 ;
11849   }
11850 }
11851 \_fp\_tmp:w {i}   {\use\_none:nnn   #50}{ 0{#2}#3{#4}#5           }
11852 \_fp\_tmp:w {ii}  {\use\_none:nn    #5 }{ 00{#2}#3{#4}#5           }
11853 \_fp\_tmp:w {iii} {\use\_none:n     #5 }{ 000{#2}#3{#4}#5           }
11854 \_fp\_tmp:w {iv}  {                #5 }{ {0000}#2{#3}#4 #5       }
11855 \_fp\_tmp:w {v}   {\use\_none:nnn   #4#5 }{ 0{0000}#2{#3}#4 #5       }
11856 \_fp\_tmp:w {vi}  {\use\_none:nn    #4#5 }{ 00{0000}#2{#3}#4 #5       }
11857 \_fp\_tmp:w {vii} {\use\_none:n     #4#5 }{ 000{0000}#2{#3}#4 #5       }
11858 \_fp\_tmp:w {viii}{                #4#5 }{ {0000}0000{#2}#3 #4 #5       }
11859 \_fp\_tmp:w {ix}  {\use\_none:nnn   #3#4+#5}{ 0{0000}0000{#2}#3 #4 #5       }
11860 \_fp\_tmp:w {x}   {\use\_none:nn    #3#4+#5}{ 00{0000}0000{#2}#3 #4 #5       }
11861 \_fp\_tmp:w {xi}  {\use\_none:n     #3#4+#5}{ 000{0000}0000{#2}#3 #4 #5       }
11862 \_fp\_tmp:w {xii} {                #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5       }
11863 \_fp\_tmp:w {xiii}{\use\_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5       }
11864 \_fp\_tmp:w {xiv} {\use\_none:nn    #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5       }
11865 \_fp\_tmp:w {xv}  {\use\_none:n     #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5       }
11866 \_fp\_tmp:w {xvi} {                #2#3+#4#5}{ {0000}0000{0000}0000 #2 #3 #4 #5 }
```

(End definition for `_fp_decimate_auxi:Nnnnn` and others.)

⁸No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

`_fp_decimate_pack:nnnnnnnnnw` The computation of the *<rounding>* digit leaves an unfinished `\int_value:w`, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a semicolon.

```

11867 \cs_new:Npn \_fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
11868 { \_fp_decimate_pack:nnnnnw { #1#2#3#4#5 } }
11869 \cs_new:Npn \_fp_decimate_pack:nnnnnw #1 #2#3#4#5#6
11870 { {#1} {#2#3#4#5#6} }

```

(End definition for `_fp_decimate_pack:nnnnnnnnnw`.)

22.10 Functions for use within primitive conditional branches

The functions described in this section are not pretty and can easily be misused. When correctly used, each of them removes one `\fi:` as part of its parameter text, and puts one back as part of its replacement text.

Many computation functions in `l3fp` must perform tests on the type of floating points that they receive. This is often done in an `\if_case:w` statement or another conditional statement, and only a few cases lead to actual computations: most of the special cases are treated using a few standard functions which we define now. A typical use context for those functions would be

```

\if_case:w <integer> \exp_stop_f:
  \_fp_case_return_o:Nw <fp var>
\or: \_fp_case_use:nw {<some computation>}
\or: \_fp_case_return_same_o:w
\or: \_fp_case_return:nw {<something>}
\fi:
<junk>
<floating point>

```

In this example, the case 0 returns the floating point *<fp var>*, expanding once after that floating point. Case 1 does *<some computation>* using the *<floating point>* (presumably compute the operation requested by the user in that non-trivial case). Case 2 returns the *<floating point>* without modifying it, removing the *<junk>* and expanding once after. Case 3 closes the conditional, removes the *<junk>* and the *<floating point>*, and expands *<something>* next. In other cases, the “*<junk>*” is expanded, performing some other operation on the *<floating point>*. We provide similar functions with two trailing *<floating points>*.

`_fp_case_use:nw` This function ends a TeX conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

```

11871 \cs_new:Npn \_fp_case_use:nw #1#2 \fi: #3 \s_fp { \fi: #1 \s_fp }

```

(End definition for `_fp_case_use:nw`.)

`_fp_case_return:nw` This function ends a TeX conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don’t define this function requiring a floating point to follow, simply anything ending in a semicolon. This, in turn, means that the *<junk>* may not contain semicolons.

```

11872 \cs_new:Npn \_fp_case_return:nw #1#2 \fi: #3 ; { \fi: #1 }

```

(End definition for _fp_case_return:nw.)

_fp_case_return_o:Nw This function ends a T_EX conditional, removes junk and a floating point, and returns its first argument (an *<fp var>*) then expands once after it.

```
11873 \cs_new:Npn \_fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
11874 { \fi: \exp_after:wN #1 }
```

(End definition for _fp_case_return_o:Nw.)

_fp_case_return_same_o:w This function ends a T_EX conditional, removes junk, and returns the following floating point, expanding once after it.

```
11875 \cs_new:Npn \_fp_case_return_same_o:w #1 \fi: #2 \s__fp
11876 { \fi: \_fp_exp_after_o:w \s__fp }
```

(End definition for _fp_case_return_same_o:w.)

_fp_case_return_o:Nww Same as _fp_case_return_o:Nw but with two trailing floating points.

```
11877 \cs_new:Npn \_fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
11878 { \fi: \exp_after:wN #1 }
```

(End definition for _fp_case_return_o:Nww.)

_fp_case_return_i_o:ww Similar to _fp_case_return_same_o:w, but this returns the first or second of two trailing floating point numbers, expanding once after the result.

```
11879 \cs_new:Npn \_fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
11880 { \fi: \_fp_exp_after_o:w \s__fp #3 ; }
11881 \cs_new:Npn \_fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
11882 { \fi: \_fp_exp_after_o:w }
```

(End definition for _fp_case_return_i_o:ww and _fp_case_return_ii_o:ww.)

22.11 Integer floating points

_fp_int_p:w Tests if the floating point argument is an integer. For normal floating point numbers, _fp_int:wTF this holds if the rounding digit resulting from _fp_decimate:nNnnnn is 0.

```
11883 \prg_new_conditional:Npmn \_fp_int:w \s__fp \_fp_chk:w #1 #2 #3 #4;
11884 { TF , T , F , p }
11885 {
11886   \if_case:w #1 \exp_stop_f:
11887     \prg_return_true:
11888   \or:
11889     \if_charcode:w 0
11890       \_fp_decimate:nNnnnn { \c__fp_prec_int - #3 }
11891       \_fp_use_i_until_s:nw #4
11892       \prg_return_true:
11893     \else:
11894       \prg_return_false:
11895     \fi:
11896   \else: \prg_return_false:
11897   \fi:
11898 }
```

(End definition for _fp_int:wTF.)

22.12 Small integer floating points

```

\__fp_small_int:wTF
\__fp_small_int_true:wTF
\__fp_small_int_normal:NnwTF
\__fp_small_int_test:NnnwNTF

```

Tests if the floating point argument is an integer or $\pm\infty$. If so, it is converted to an integer in the range $[-10^8, 10^8]$ and fed as a braced argument to the *⟨true code⟩*. Otherwise, the *⟨false code⟩* is performed.

First filter special cases: zeros and infinities are integers, `nan` is not. For normal numbers, decimate. If the rounding digit is not 0 run the *⟨false code⟩*. If it is, then the integer is #2 #3; use #3 if #2 vanishes and otherwise 10^8 .

```

11899 \cs_new:Npn \__fp_small_int:wTF \s__fp \__fp_chk:w #1#2
11900 {
11901   \if_case:w #1 \exp_stop_f:
11902     \__fp_case_return:nw { \__fp_small_int_true:wTF 0 ; }
11903   \or:   \exp_after:wN \__fp_small_int_normal:NnwTF
11904   \or:
11905     \__fp_case_return:nw
11906     {
11907       \exp_after:wN \__fp_small_int_true:wTF \int_value:w
11908       \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
11909     }
11910   \else: \__fp_case_return:nw \use_ii:nn
11911   \fi:
11912   #2
11913 }
11914 \cs_new:Npn \__fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
11915 \cs_new:Npn \__fp_small_int_normal:NnwTF #1#2#3;
11916 {
11917   \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
11918   \__fp_small_int_test:NnnwNw
11919   #3 #1
11920 }
11921 \cs_new:Npn \__fp_small_int_test:NnnwNw #1#2#3#4; #5
11922 {
11923   \if_meaning:w 0 #1
11924     \exp_after:wN \__fp_small_int_true:wTF
11925     \int_value:w \if_meaning:w 2 #5 - \fi:
11926     \if_int_compare:w #2 > 0 \exp_stop_f:
11927     1 0000 0000
11928   \else:
11929     #3
11930   \fi:
11931   \exp_after:wN ;
11932 \else:
11933   \exp_after:wN \use_ii:nn
11934 \fi:
11935 }

```

(End definition for `__fp_small_int:wTF` and others.)

22.13 x-like expansion expandably

```

\__fp_expand:n
\__fp_expand_loop:nwnN

```

This expandable function behaves in a way somewhat similar to `\use:x`, but much less robust. The argument is f-expanded, then the leading item (often a single character token) is moved to a storage area after `\s__fp_mark`, and f-expansion is applied again, repeating until the argument is empty. The result built one piece at a time is then

inserted in the input stream. Note that spaces are ignored by this procedure, unless surrounded with braces. Multiple tokens which do not need expansion can be inserted within braces.

```

11936 \cs_new:Npn \__fp_expand:n #1
11937 {
11938   \__fp_expand_loop:nwnN { }
11939   #1 \prg_do_nothing:
11940   \s__fp_mark { } \__fp_expand_loop:nwnN
11941   \s__fp_mark { } \__fp_use_i_until_s:nw ;
11942 }
11943 \cs_new:Npn \__fp_expand_loop:nwnN #1#2 \s__fp_mark #3 #4
11944 {
11945   \exp_after:wN #4 \exp:w \exp_end_continue_f:w
11946   #2
11947   \s__fp_mark { #3 #1 } #4
11948 }

```

(End definition for __fp_expand:n and __fp_expand_loop:nwnN.)

22.14 Fast string comparison

`__fp_str_if_eq_x:nn` A private version of the low-level string comparison function. As the nature of the arguments is restricted and as speed is of the essence, this version does not seek to deal with `#` tokens. No `l3sys` or `l3luatex` just yet so we have to define in terms of primitives.

```

11949 \cs_new:Npn \__fp_str_if_eq_x:nn #1#2 { \pdfstrcmp:D {#1} {#2} }
11950 \sys_if_engine_luatex:T
11951 {
11952   \cs_set:Npn \__fp_str_if_eq_x:nn #1#2
11953   {
11954     \luatex_directlua:D
11955     {
11956       l3kernel_strcmp
11957       (
11958         " \luatex_luaescapestring:D {#1}",
11959         " \luatex_luaescapestring:D {#2}"
11960       )
11961     }
11962   }
11963 }

```

(End definition for __fp_str_if_eq_x:nn.)

22.15 Name of a function from its `l3fp`-parse name

`__fp_func_to_name:N` The goal is to convert for instance `__fp_sin_o:w` to `sin`. This is used in error messages
`__fp_func_to_name_aux:w` hence does not need to be fast.

```

11964 \cs_new:Npn \__fp_func_to_name:N #1
11965 {
11966   \exp_last_unbraced:Nf
11967   \__fp_func_to_name_aux:w { \cs_to_str:N #1 } X
11968 }
11969 \cs_set_protected:Npn \__fp_tmp:w #1 #2
11970 { \cs_new:Npn \__fp_func_to_name_aux:w ##1 #1 ##2 #2 ##3 X {##2} }

```

```

11971 \exp_args:Nff \__fp_tmp:w { \tl_to_str:n { __fp_ } }
11972 { \tl_to_str:n { _o: } }

```

(End definition for __fp_func_to_name:N and __fp_func_to_name_aux:w.)

22.16 Messages

Using a floating point directly is an error.

```

11973 \__kernel_msg_new:nnnn { kernel } { misused-fp }
11974 { A~floating~point~with~value~'#1'~was~misused. }
11975 {
11976   To~obtain~the~value~of~a~floating~point~variable,~use~
11977   '\token_to_str:N \fp_to_decimal:N',~
11978   '\token_to_str:N \fp_to_tl:N',~or~other~
11979   conversion~functions.
11980 }
11981 </initex | package>

```

23 13fp-traps Implementation

```

11982 (*initex | package)
11983 <@@=fp>

```

Exceptions should be accessed by an n-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

23.1 Flags

`flag_fp_invalid_operation` Flags to denote exceptions.

```

flag_fp_division_by_zero 11984 \flag_new:n { fp_invalid_operation }
flag_fp_overflow         11985 \flag_new:n { fp_division_by_zero }
flag_fp_underflow        11986 \flag_new:n { fp_overflow }
                          11987 \flag_new:n { fp_underflow }

```

(End definition for `flag_fp_invalid_operation` and others. These variables are documented on page 187.)

23.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an N-type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the inexact exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- `__fp_invalid_operation:nnw`,
- `__fp_invalid_operation_o:Nww`,
- `__fp_invalid_operation_tl_o:ff`,
- `__fp_division_by_zero_o:Nnw`,
- `__fp_division_by_zero_o:NNww`,
- `__fp_overflow:w`,
- `__fp_underflow:w`.

Rather than changing them directly, we provide a user interface as `\fp_trap:nn` $\{\langle exception \rangle\}$ $\{\langle way of trapping \rangle\}$, where the $\langle way of trapping \rangle$ is one of `error`, `flag`, or `none`.

We also provide `__fp_invalid_operation_o:nw`, defined in terms of `__fp_invalid_operation:nnw`.

`\fp_trap:nn`

```

11988 \cs_new_protected:Npn \fp_trap:nn #1#2
11989 {
11990   \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
11991   {
11992     \clist_if_in:nnTF
11993       { invalid_operation , division_by_zero , overflow , underflow }
11994       {#1}
11995       {
11996         \__kernel_msg_error:nnxx { kernel }
11997           { unknown-fpu-trap-type } {#1} {#2}
11998       }
11999       {
12000         \__kernel_msg_error:nnx
12001           { kernel } { unknown-fpu-exception } {#1}
12002       }
12003   }
12004 }
```

(End definition for `\fp_trap:nn`. This function is documented on page 187.)

`_fp_trap_invalid_operation_set_error:` We provide three types of trapping for invalid operations: either produce an error and
`_fp_trap_invalid_operation_set_flag:` raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases,
`_fp_trap_invalid_operation_set_none:` the function produces as a result its first argument, possibly with post-expansion.
`_fp_trap_invalid_operation_set:N`

```

12005 \cs_new_protected:Npn \_fp_trap_invalid_operation_set_error:
12006 { \_fp_trap_invalid_operation_set:N \prg_do_nothing: }
12007 \cs_new_protected:Npn \_fp_trap_invalid_operation_set_flag:
12008 { \_fp_trap_invalid_operation_set:N \use_none:nnnnn }
12009 \cs_new_protected:Npn \_fp_trap_invalid_operation_set_none:
12010 { \_fp_trap_invalid_operation_set:N \use_none:nnnnnnnn }
12011 \cs_new_protected:Npn \_fp_trap_invalid_operation_set:N #1
12012 {
12013   \exp_args:Nno \use:n
12014   { \cs_set:Npn \_fp_invalid_operation:nnw ##1##2##3; }
12015   {
12016     #1
12017     \_fp_error:nnfn { fp-invalid } {##2} { \fp_to_tl:n { ##3; } } { }
12018     \flag_raise_if_clear:n { fp_invalid_operation }
12019     ##1
12020   }
12021   \exp_args:Nno \use:n
12022   { \cs_set:Npn \_fp_invalid_operation_o:Nww ##1##2; ##3; }
12023   {
12024     #1
12025     \_fp_error:nffn { fp-invalid-ii }
12026     { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
12027     \flag_raise_if_clear:n { fp_invalid_operation }
12028     \exp_after:wN \c_nan_fp
12029   }
12030   \exp_args:Nno \use:n
12031   { \cs_set:Npn \_fp_invalid_operation_tl_o:ff ##1##2 }
12032   {
12033     #1
12034     \_fp_error:nffn { fp-invalid } {##1} {##2} { }
12035     \flag_raise_if_clear:n { fp_invalid_operation }
12036     \exp_after:wN \c_nan_fp
12037   }
12038 }

```

(End definition for `_fp_trap_invalid_operation_set_error:` and others.)

`_fp_trap_division_by_zero_set_error:` We provide three types of trapping for invalid operations and division by zero: either
`_fp_trap_division_by_zero_set_flag:` produce an error and raise the relevant flag; or only raise the flag; or don't even raise the
`_fp_trap_division_by_zero_set_none:` flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or
`_fp_trap_division_by_zero_set:N` NaN.

```

12039 \cs_new_protected:Npn \_fp_trap_division_by_zero_set_error:
12040 { \_fp_trap_division_by_zero_set:N \prg_do_nothing: }
12041 \cs_new_protected:Npn \_fp_trap_division_by_zero_set_flag:
12042 { \_fp_trap_division_by_zero_set:N \use_none:nnnnn }
12043 \cs_new_protected:Npn \_fp_trap_division_by_zero_set_none:
12044 { \_fp_trap_division_by_zero_set:N \use_none:nnnnnnnn }
12045 \cs_new_protected:Npn \_fp_trap_division_by_zero_set:N #1
12046 {
12047   \exp_args:Nno \use:n
12048   { \cs_set:Npn \_fp_division_by_zero_o:Nnw ##1##2##3; }

```

```

12049     {
12050         #1
12051         \_fp_error:nfn { fp-zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
12052         \flag_raise_if_clear:n { fp_division_by_zero }
12053         \exp_after:wN ##1
12054     }
12055     \exp_args:Nno \use:n
12056     { \cs_set:Npn \_fp_division_by_zero_o:NNww ##1##2##3; ##4; }
12057     {
12058         #1
12059         \_fp_error:nfn { fp-zero-div-ii }
12060         { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
12061         \flag_raise_if_clear:n { fp_division_by_zero }
12062         \exp_after:wN ##1
12063     }
12064 }

```

(End definition for `_fp_trap_division_by_zero_set_error:` and others.)

`_fp_trap_overflow_set_error:` Just as for invalid operations and division by zero, the three different behaviours are obtained by feeding `\prg_do_nothing:`, `\use_none:nnnnn` or `\use_none:nnnnnnn` to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the `_fp_overflow:w` and `_fp_underflow:w` functions will be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as $10 \cdot 10^{9999}$, the exponent would be too large for T_EX, and `_fp_overflow:w` receives $\pm\infty$ (`_fp_underflow:w` would receive ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```

12065 \cs_new_protected:Npn \_fp_trap_overflow_set_error:
12066 { \_fp_trap_overflow_set:N \prg_do_nothing: }
12067 \cs_new_protected:Npn \_fp_trap_overflow_set_flag:
12068 { \_fp_trap_overflow_set:N \use_none:nnnnn }
12069 \cs_new_protected:Npn \_fp_trap_overflow_set_none:
12070 { \_fp_trap_overflow_set:N \use_none:nnnnnnn }
12071 \cs_new_protected:Npn \_fp_trap_overflow_set:N #1
12072 { \_fp_trap_overflow_set:NnNn #1 { overflow } \_fp_inf_fp:N { inf } }
12073 \cs_new_protected:Npn \_fp_trap_underflow_set_error:
12074 { \_fp_trap_underflow_set:N \prg_do_nothing: }
12075 \cs_new_protected:Npn \_fp_trap_underflow_set_flag:
12076 { \_fp_trap_underflow_set:N \use_none:nnnnn }
12077 \cs_new_protected:Npn \_fp_trap_underflow_set_none:
12078 { \_fp_trap_underflow_set:N \use_none:nnnnnnn }
12079 \cs_new_protected:Npn \_fp_trap_underflow_set:N #1
12080 { \_fp_trap_overflow_set:NnNn #1 { underflow } \_fp_zero_fp:N { 0 } }
12081 \cs_new_protected:Npn \_fp_trap_overflow_set:NnNn #1#2#3#4
12082 {
12083     \exp_args:Nno \use:n
12084     { \cs_set:cpn { \_fp_ #2 :w } \s_fp \_fp_chk:w ##1##2##3; }
12085     {
12086         #1
12087         \_fp_error:nfn
12088         { fp-flow \if_meaning:w 1 ##1 -to \fi: }
12089         { \fp_to_tl:n { \s_fp \_fp_chk:w ##1##2##3; } }
12090         { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }

```

```

12091         {#2}
12092         \flag_raise_if_clear:n { fp_#2 }
12093         #3 ##2
12094     }
12095 }

```

(End definition for `__fp_trap_overflow_set_error:` and others.)

`__fp_invalid_operation:nnw` Initialize the control sequences (to log properly their existence). Then set invalid operations to trigger an error, and division by zero, overflow, and underflow to act silently on their flag.

```

\__fp_invalid_operation_o:Nww
\__fp_invalid_operation_tl_o:ff
\__fp_division_by_zero_o:Nnw
\__fp_division_by_zero_o:NNww
\__fp_overflow:w
\__fp_underflow:w
12096 \cs_new:Npn \__fp_invalid_operation:nnw #1#2#3; { }
12097 \cs_new:Npn \__fp_invalid_operation_o:Nww #1#2; #3; { }
12098 \cs_new:Npn \__fp_invalid_operation_tl_o:ff #1 #2 { }
12099 \cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3; { }
12100 \cs_new:Npn \__fp_division_by_zero_o:NNww #1#2#3; #4; { }
12101 \cs_new:Npn \__fp_overflow:w { }
12102 \cs_new:Npn \__fp_underflow:w { }
12103 \fp_trap:nn { invalid_operation } { error }
12104 \fp_trap:nn { division_by_zero } { flag }
12105 \fp_trap:nn { overflow } { flag }
12106 \fp_trap:nn { underflow } { flag }

```

(End definition for `__fp_invalid_operation:nnw` and others.)

`__fp_invalid_operation_o:nw` Convenient short-hands for returning `\c_nan_fp` for a unary or binary operation, and `__fp_invalid_operation_o:fw` expanding after.

```

12107 \cs_new:Npn \__fp_invalid_operation_o:nw
12108 { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
12109 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }

```

(End definition for `__fp_invalid_operation_o:nw`.)

23.3 Errors

```

\__fp_error:nnnn
\__fp_error:nnfn
\__fp_error:nffn
\__fp_error:nfff
12110 \cs_new:Npn \__fp_error:nnnn
12111 { \__kernel_msg_expandable_error:nnnnn { kernel } }
12112 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff , nfff }

```

(End definition for `__fp_error:nnnn`.)

23.4 Messages

Some messages.

```

12113 \__kernel_msg_new:nnnn { kernel } { unknown-fpu-exception }
12114 {
12115     The-FPU-exception-~'~#1'~is~not~known:~
12116     that~trap~will~never~be~triggered.
12117 }
12118 {
12119     The-only-exceptions~to~which~traps~can~be~attached~are \
12120     \iow_indent:n
12121     {

```

```

12122         * ~ invalid_operation \\
12123         * ~ division_by_zero \\
12124         * ~ overflow \\
12125         * ~ underflow
12126     }
12127 }
12128 \__kernel_msg_new:nnnn { kernel } { unknown-fpu-trap-type }
12129 { The-FPU-trap-type-~'~#2'~is~not~known. }
12130 {
12131     The-trap-type-must-be-one-of \\
12132     \iow_indent:n
12133     {
12134         * ~ error \\
12135         * ~ flag \\
12136         * ~ none
12137     }
12138 }
12139 \__kernel_msg_new:nnn { kernel } { fp-flow }
12140 { An ~ #3 ~ occurred. }
12141 \__kernel_msg_new:nnn { kernel } { fp-flow-to }
12142 { #1 ~ #3 ed ~ to ~ #2 . }
12143 \__kernel_msg_new:nnn { kernel } { fp-zero-div }
12144 { Division-by-zero-in~ #1 (#2) }
12145 \__kernel_msg_new:nnn { kernel } { fp-zero-div-ii }
12146 { Division-by-zero-in~ (#1) #3 (#2) }
12147 \__kernel_msg_new:nnn { kernel } { fp-invalid }
12148 { Invalid-operation~ #1 (#2) }
12149 \__kernel_msg_new:nnn { kernel } { fp-invalid-ii }
12150 { Invalid-operation~ (#1) #3 (#2) }
12151 \__kernel_msg_new:nnn { kernel } { fp-unknown-type }
12152 { Unknown-type-for~'~#1' }
12153 </initex | package>

```

24 I3fp-round implementation

```

12154 (*initex | package)
12155 <@@=fp>

\__fp_parse_word_trunc:N
\__fp_parse_word_floor:N
\__fp_parse_word_ceil:N
12156 \cs_new:Npn \__fp_parse_word_trunc:N
12157 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_zero:NNN }
12158 \cs_new:Npn \__fp_parse_word_floor:N
12159 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_ninf:NNN }
12160 \cs_new:Npn \__fp_parse_word_ceil:N
12161 { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_pinf:NNN }

(End definition for \__fp_parse_word_trunc:N, \__fp_parse_word_floor:N, and \__fp_parse_word_ceil:N.)

\__fp_parse_word_round:N
\__fp_parse_round:Nw
12162 \cs_new:Npn \__fp_parse_word_round:N #1#2
12163 {
12164     \__fp_parse_function:NNN

```

```

12165     \__fp_round_o:Nw \__fp_round_to_nearest:NNN #1
12166     #2
12167   }
12168 \cs_new:Npn \__fp_parse_round:Nw #1 #2 \__fp_round_to_nearest:NNN #3#4
12169   { #2 #1 #3 }
12170

```

(End definition for `__fp_parse_word_round:N` and `__fp_parse_round:Nw`.)

24.1 Rounding tools

`\c__fp_five_int` This is used as the half-point for which numbers are rounded up/down.

```

12171 \int_const:Nn \c__fp_five_int { 5 }

```

(End definition for `\c__fp_five_int`.)

Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- Round to nearest: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.
- Round towards negative infinity: round to the greatest floating point number not larger than the exact result.
- Round towards zero: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.
- Round towards positive infinity: round to the least floating point number not smaller than the exact result.

This is not fully implemented in `l3fp` yet, and transcendental functions fall back on the “round to nearest” mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behaviour (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function `__fp_round:NNN`, which expands to `0\exp_stop_f:` or `1\exp_stop_f:` depending on whether the final result should be rounded up or down.

- `__fp_round:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f:`.
- `__fp_round_s:NNNw <sign> <digit1> <digit2> <more digits>`; can expand to `0\exp_stop_f;` or `1\exp_stop_f;`.
- `__fp_round_neg:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f:`.

See implementation comments for details on the syntax.

```

    \_fp\_round:NNN
\_fp\_round\_to\_nearest:NNN
    \_fp\_round\_to\_nearest\_ninf:NNN
    \_fp\_round\_to\_nearest\_zero:NNN
    \_fp\_round\_to\_nearest\_pinf:NNN
\_fp\_round\_to\_ninf:NNN
\_fp\_round\_to\_zero:NNN
\_fp\_round\_to\_pinf:NNN

```

```

    \_fp\_round:NNN  $\langle final\ sign \rangle$   $\langle digit_1 \rangle$   $\langle digit_2 \rangle$ 

```

If rounding the number $\langle final\ sign \rangle \langle digit_1 \rangle . \langle digit_2 \rangle$ to an integer rounds it towards zero (truncates it), this function expands to `0\exp_stop_f:`, and otherwise to `1\exp_stop_f:`. Typically used within the scope of an `_fp_int_eval:w`, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that $\langle final\ sign \rangle$ be the final sign of the result. Otherwise, the result would be incorrect in the case of rounding towards $-\infty$ or towards $+\infty$. Also recall that $\langle final\ sign \rangle$ is 0 for positive, and 2 for negative.

By default, the functions below return `0\exp_stop_f:`, but this is superseded by `_fp_round_return_one:`, which instead returns `1\exp_stop_f:`, expanding everything and removing `0\exp_stop_f:` in the process. In the case of rounding towards $\pm\infty$ or towards 0, this is not really useful, but it prepares us for the “round to nearest, ties to even” mode.

The “round to nearest” mode is the default. If the $\langle digit_2 \rangle$ is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that $\langle digit_1 \rangle$ plus the result is even. In other words, round up if $\langle digit_1 \rangle$ is odd.

The “round to nearest” mode has three variants, which differ in how ties are rounded: down towards $-\infty$, truncated towards 0, or up towards $+\infty$.

```

12172 \cs\_new:Npn \_fp\_round\_return\_one:
12173 { \exp\_after:wN 1 \exp\_after:wN \exp\_stop\_f: \exp:w }
12174 \cs\_new:Npn \_fp\_round\_to\_ninf:NNN #1 #2 #3
12175 {
12176   \if\_meaning:w 2 #1
12177     \if\_int\_compare:w #3 > 0 \exp\_stop\_f:
12178       \_fp\_round\_return\_one:
12179     \fi:
12180   \fi:
12181   0 \exp\_stop\_f:
12182 }
12183 \cs\_new:Npn \_fp\_round\_to\_zero:NNN #1 #2 #3 { 0 \exp\_stop\_f: }
12184 \cs\_new:Npn \_fp\_round\_to\_pinf:NNN #1 #2 #3
12185 {
12186   \if\_meaning:w 0 #1
12187     \if\_int\_compare:w #3 > 0 \exp\_stop\_f:
12188       \_fp\_round\_return\_one:
12189     \fi:
12190   \fi:
12191   0 \exp\_stop\_f:
12192 }
12193 \cs\_new:Npn \_fp\_round\_to\_nearest:NNN #1 #2 #3
12194 {
12195   \if\_int\_compare:w #3 > \c\_fp\_five\_int
12196     \_fp\_round\_return\_one:
12197   \else:
12198     \if\_meaning:w 5 #3
12199       \if\_int\_odd:w #2 \exp\_stop\_f:
12200       \_fp\_round\_return\_one:
12201     \fi:
12202   \fi:
12203   \fi:
12204   0 \exp\_stop\_f:
12205 }

```

```

12206 \cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
12207 {
12208   \if_int_compare:w #3 > \c__fp_five_int
12209     \__fp_round_return_one:
12210   \else:
12211     \if_meaning:w 5 #3
12212       \if_meaning:w 2 #1
12213         \__fp_round_return_one:
12214       \fi:
12215     \fi:
12216   \fi:
12217   0 \exp_stop_f:
12218 }
12219 \cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
12220 {
12221   \if_int_compare:w #3 > \c__fp_five_int
12222     \__fp_round_return_one:
12223   \fi:
12224   0 \exp_stop_f:
12225 }
12226 \cs_new:Npn \__fp_round_to_nearest_pinf:NNN #1 #2 #3
12227 {
12228   \if_int_compare:w #3 > \c__fp_five_int
12229     \__fp_round_return_one:
12230   \else:
12231     \if_meaning:w 5 #3
12232       \if_meaning:w 0 #1
12233         \__fp_round_return_one:
12234       \fi:
12235     \fi:
12236   \fi:
12237   0 \exp_stop_f:
12238 }
12239 \cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN

```

(End definition for __fp_round:NNN and others.)

__fp_round_s:NNNw __fp_round_s:NNNw <final sign> <digit> <more digits> ;

Similar to __fp_round:NNN, but with an extra semicolon, this function expands to 0\exp_stop_f:; if rounding <final sign><digit>.<more digits> to an integer truncates, and to 1\exp_stop_f:; otherwise. The <more digits> part must be a digit, followed by something that does not overflow a \int_use:N __fp_int_eval:w construction. The only relevant information about this piece is whether it is zero or not.

```

12240 \cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
12241 {
12242   \exp_after:wN \__fp_round:NNN
12243   \exp_after:wN #1
12244   \exp_after:wN #2
12245   \int_value:w \__fp_int_eval:w
12246   \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
12247   \if_meaning:w 5 #3 1 \fi:
12248   \exp_stop_f:
12249   \if_int_compare:w \__fp_int_eval:w #4 > 0 \exp_stop_f:
12250   1 +

```

```

12251         \fi:
12252     \fi:
12253     #3
12254 ;
12255 }

```

(End definition for `__fp_round_s:NNNw`.)

`__fp_round_digit:Nw`

```

\int_value:w \__fp_round_digit:Nw <digit> <intexpr> ;

```

This function should always be called within an `\int_value:w` or `__fp_int_eval:w` expansion; it may add an extra `__fp_int_eval:w`, which means that the integer or integer expression should not be ended with a synonym of `\relax`, but with a semi-colon for instance.

```

12256 \cs_new:Npn \__fp_round_digit:Nw #1 #2;
12257 {
12258     \if_int_odd:w \if_meaning:w 0 #1 1 \else:
12259         \if_meaning:w 5 #1 1 \else:
12260         0 \fi: \fi: \exp_stop_f:
12261     \if_int_compare:w \__fp_int_eval:w #2 > 0 \exp_stop_f:
12262     \__fp_int_eval:w 1 +
12263     \fi:
12264     \fi:
12265     #1
12266 }

```

(End definition for `__fp_round_digit:Nw`.)

`__fp_round_neg:NNN`

```

\__fp_round_neg:NNN <final sign> <digit1> <digit2>

```

`_fp_round_to_nearest_neg:NNN`

This expands to `0\exp_stop_f:` or `1\exp_stop_f:` after doing the following test.

`_fp_round_to_nearest_ninf_neg:NNN`

Starting from a number of the form $\langle final\ sign \rangle 0.\langle 15\ digits \rangle \langle digit_1 \rangle$ with exactly 15 (non-all-zero) digits before $\langle digit_1 \rangle$, subtract from it $\langle final\ sign \rangle 0.0\dots 0 \langle digit_2 \rangle$, where there are 16 zeros. If in the current rounding mode the result should be rounded down, then this function returns `1\exp_stop_f:`. Otherwise, *i.e.*, if the result is rounded back to the first operand, then this function returns `0\exp_stop_f:`.

`_fp_round_to_nearest_zero_neg:NNN`

`_fp_round_to_nearest_pinf_neg:NNN`

`__fp_round_to_ninf_neg:NNN`

`__fp_round_to_zero_neg:NNN`

`__fp_round_to_pinf_neg:NNN`

It turns out that this negative “round to nearest” is identical to the positive one. And this is the default mode.

```

12267 \cs_new_eq:NN \__fp_round_to_ninf_neg:NNN \__fp_round_to_pinf:NNN
12268 \cs_new:Npn \__fp_round_to_zero_neg:NNN #1 #2 #3
12269 {
12270     \if_int_compare:w #3 > 0 \exp_stop_f:
12271     \__fp_round_return_one:
12272     \fi:
12273     0 \exp_stop_f:
12274 }
12275 \cs_new_eq:NN \__fp_round_to_pinf_neg:NNN \__fp_round_to_ninf:NNN
12276 \cs_new_eq:NN \__fp_round_to_nearest_neg:NNN \__fp_round_to_nearest:NNN
12277 \cs_new_eq:NN \__fp_round_to_nearest_ninf_neg:NNN
12278     \__fp_round_to_nearest_pinf:NNN
12279 \cs_new:Npn \__fp_round_to_nearest_zero_neg:NNN #1 #2 #3
12280 {
12281     \if_int_compare:w #3 < \c__fp_five_int \else:
12282     \__fp_round_return_one:
12283     \fi:

```



```

12284     0 \exp_stop_f:
12285   }
12286 \cs_new_eq:NN \__fp_round_to_nearest_pinf_neg:NNN
12287   \__fp_round_to_nearest_ninf:NNN
12288 \cs_new_eq:NN \__fp_round_neg:NNN \__fp_round_to_nearest_neg:NNN

```

(End definition for __fp_round_neg:NNN and others.)

24.2 The round function

__fp_round_o:Nw First check that all arguments are floating point numbers. The `trunc`, `ceil` and `floor` functions expect one or two arguments (the second is 0 by default), and the `round` function also accepts a third argument (`nan` by default), which changes `#1` from `__fp_round_to_nearest:NNN` to one of its analogues.

```

12289 \cs_new:Npn \__fp_round_o:Nw #1
12290 {
12291   \__fp_parse_function_all_fp_o:fnw
12292   { \__fp_round_name_from_cs:N #1 }
12293   { \__fp_round_aux_o:Nw #1 }
12294 }
12295 \cs_new:Npn \__fp_round_aux_o:Nw #1#2 @
12296 {
12297   \if_case:w
12298     \__fp_int_eval:w \__fp_array_count:n {#2} \__fp_int_eval_end:
12299     \__fp_round_no_arg_o:Nw #1 \exp:w
12300   \or: \__fp_round:Nwn #1 #2 {0} \exp:w
12301   \or: \__fp_round:Nww #1 #2 \exp:w
12302   \else: \__fp_round:Nwww #1 #2 @ \exp:w
12303   \fi:
12304   \exp_after:wN \exp_end:
12305 }

```

(End definition for __fp_round_o:Nw and __fp_round_aux_o:Nw.)

__fp_round_no_arg_o:Nw

```

12306 \cs_new:Npn \__fp_round_no_arg_o:Nw #1
12307 {
12308   \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
12309   { \__fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 } }
12310   {
12311     \__fp_error:nffn { fp-num-args }
12312     { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
12313   }
12314   \exp_after:wN \c_nan_fp
12315 }

```

(End definition for __fp_round_no_arg_o:Nw.)

__fp_round:Nwww Having three arguments is only allowed for `round`, not `trunc`, `ceil`, `floor`, so check for that case. If all is well, construct one of `__fp_round_to_nearest:NNN`, `__fp_round_to_nearest_zero:NNN`, `__fp_round_to_nearest_ninf:NNN`, `__fp_round_to_nearest_pinf:NNN` and act accordingly.

```

12316 \cs_new:Npn \__fp_round:Nwww #1#2 ; #3 ; \s__fp \__fp_chk:w #4#5#6 ; #7 @
12317 {

```

```

12318 \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
12319 {
12320   \tl_if_empty:nTF {#7}
12321   {
12322     \exp_args:Nc \__fp_round:Nww
12323     {
12324       __fp_round_to_nearest
12325       \if_meaning:w 0 #4 _zero \else:
12326       \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
12327       :NNN
12328     }
12329     #2 ; #3 ;
12330   }
12331   {
12332     \__fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 }
12333     \exp_after:wN \c_nan_fp
12334   }
12335 }
12336 {
12337   \__fp_error:nffn { fp-num-args }
12338   { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
12339   \exp_after:wN \c_nan_fp
12340 }
12341 }

```

(End definition for __fp_round:Nwww.)

__fp_round_name_from_cs:N

```

12342 \cs_new:Npn \__fp_round_name_from_cs:N #1
12343 {
12344   \cs_if_eq:NNTF #1 \__fp_round_to_zero:NNN { trunc }
12345   {
12346     \cs_if_eq:NNTF #1 \__fp_round_to_ninf:NNN { floor }
12347     {
12348       \cs_if_eq:NNTF #1 \__fp_round_to_pinf:NNN { ceil }
12349       { round }
12350     }
12351   }
12352 }

```

(End definition for __fp_round_name_from_cs:N.)

```

\__fp_round:Nww
\__fp_round:Nwn
\__fp_round_normal:NwNNnw
\__fp_round_normal:NnnwNNnn
\__fp_round_pack:Nw
\__fp_round_normal:NNwNnn
\__fp_round_normal_end:wwNnn
\__fp_round_special:NwwNnn
\__fp_round_special_aux:Nw
12353 \cs_new:Npn \__fp_round:Nww #1#2 ; #3 ;
12354 {
12355   \__fp_small_int:wTF #3; { \__fp_round:Nwn #1#2; }
12356   {
12357     \__fp_invalid_operation_tl_o:ff
12358     { \__fp_round_name_from_cs:N #1 }
12359     { \__fp_array_to_clist:n { #2; #3; } }
12360   }
12361 }
12362 \cs_new:Npn \__fp_round:Nwn #1 \s__fp \__fp_chk:w #2#3#4; #5
12363 {
12364   \if_meaning:w 1 #2

```

```

12365     \exp_after:wN \__fp_round_normal:NwNNnw
12366     \exp_after:wN #1
12367     \int_value:w #5
12368     \else:
12369     \exp_after:wN \__fp_exp_after_o:w
12370     \fi:
12371     \s__fp \__fp_chk:w #2#3#4;
12372   }
12373   \cs_new:Npn \__fp_round_normal:NwNNnw #1#2 \s__fp \__fp_chk:w 1#3#4#5;
12374   {
12375     \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 - #2 }
12376     \__fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
12377   }
12378   \cs_new:Npn \__fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
12379   {
12380     \exp_after:wN \__fp_round_normal:NNwNnn
12381     \int_value:w \__fp_int_eval:w
12382     \if_int_compare:w #2 > 0 \exp_stop_f:
12383       1 \int_value:w #2
12384       \exp_after:wN \__fp_round_pack:Nw
12385       \int_value:w \__fp_int_eval:w 1#3 +
12386     \else:
12387       \if_int_compare:w #3 > 0 \exp_stop_f:
12388         1 \int_value:w #3 +
12389       \fi:
12390     \fi:
12391     \exp_after:wN #5
12392     \exp_after:wN #6
12393     \use_none:nnnnnnn #3
12394     #1
12395     \__fp_int_eval_end:
12396     0000 0000 0000 0000 ; #6
12397   }
12398   \cs_new:Npn \__fp_round_pack:Nw #1
12399   { \if_meaning:w 2 #1 + 1 \fi: \__fp_int_eval_end: }
12400   \cs_new:Npn \__fp_round_normal:NNwNnn #1 #2
12401   {
12402     \if_meaning:w 0 #2
12403       \exp_after:wN \__fp_round_special:NwwNnn
12404       \exp_after:wN #1
12405     \fi:
12406     \__fp_pack_twice_four:wNNNNNNNN
12407     \__fp_pack_twice_four:wNNNNNNNN
12408     \__fp_round_normal_end:wwNnn
12409     ; #2
12410   }
12411   \cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
12412   {
12413     \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
12414     \__fp_sanitise:Nw #3 #4 ; #1 ;
12415   }
12416   \cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
12417   {
12418     \if_meaning:w 0 #1

```

```

12419     \__fp_case_return:nw
12420     { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
12421 \else:
12422     \exp_after:wN \__fp_round_special_aux:Nw
12423     \exp_after:wN #4
12424     \int_value:w \__fp_int_eval:w 1
12425     \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
12426 \fi:
12427 ;
12428 }
12429 \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
12430 {
12431     \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
12432     \__fp_sanitise:Nw #1#2; {1000}{0000}{0000}{0000};
12433 }

(End definition for \__fp_round:Nww and others.)

12434 \</initex | package>

```

25 l3fp-parse implementation

```

12435 (*initex | package)
12436 <@@=fp>

```

25.1 Work plan

The task at hand is non-trivial, and some previous failed attempts show that the code leads to unreadable logs, so we had better get it (almost) right the first time. Let us first describe our goal, then discuss the design precisely before writing any code.

In this file at least, a *<floating point object>* is a floating point number or tuple. This can be extended to anything that starts with `\s__fp` or `\s__fp_<type>` and ends with `;` with some internal structure that depends on the *<type>*.

```
\__fp_parse:n \__fp_parse:n {<fpexpr>}
```

Evaluates the *<floating point expression>* and leaves the result in the input stream as a floating point object. This function forms the basis of almost all public `l3fp` functions. During evaluation, each token is fully `f`-expanded.

`__fp_parse_o:n` does the same but expands once after its result.

T_EXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as `\int_use:N`. Invalid tokens remaining after `f`-expansion lead to unrecoverable low-level T_EX errors.

(End definition for `__fp_parse:n`.)

```

\c__fp_prec_func_int
\c__fp_prec_hatii_int
\c__fp_prec_hat_int
\c__fp_prec_not_int
\c__fp_prec_times_int
\c__fp_prec_plus_int
\c__fp_prec_comp_int
\c__fp_prec_and_int
\c__fp_prec_or_int
\c__fp_prec_quest_int
\c__fp_prec_colon_int
\c__fp_prec_comma_int
\c__fp_prec_tuple_int
\c__fp_prec_end_int

```

Floating point expressions are composed of numbers, given in various forms, infix operators, such as `+`, `**`, or `,` (which joins two numbers into a list), and prefix operators, such as the unary `-`, functions, or opening parentheses. Here is a list of precedences which control the order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals), from the tightest binding to the loosest binding.

16 Function calls.

- 13/14 Binary ****** and **^** (right to left).
- 12 Unary **+**, **-**, **!** (right to left).
- 10 Binary *****, **/**, and juxtaposition (implicit *****).
- 9 Binary **+** and **-**.
- 7 Comparisons.
- 6 Logical **and**, denoted by **&&**.
- 5 Logical **or**, denoted by **||**.
- 4 Ternary operator **?:**, piece **?**.
- 3 Ternary operator **?:**, piece **:**.
- 2 Commas.
- 1 Place where a comma is allowed and generates a tuple.
- 0 Start and end of the expression.

```

12437 \int_const:Nn \c__fp_prec_func_int    { 16 }
12438 \int_const:Nn \c__fp_prec_hatii_int   { 14 }
12439 \int_const:Nn \c__fp_prec_hat_int     { 13 }
12440 \int_const:Nn \c__fp_prec_not_int      { 12 }
12441 \int_const:Nn \c__fp_prec_times_int    { 10 }
12442 \int_const:Nn \c__fp_prec_plus_int     { 9 }
12443 \int_const:Nn \c__fp_prec_comp_int     { 7 }
12444 \int_const:Nn \c__fp_prec_and_int      { 6 }
12445 \int_const:Nn \c__fp_prec_or_int       { 5 }
12446 \int_const:Nn \c__fp_prec_quest_int    { 4 }
12447 \int_const:Nn \c__fp_prec_colon_int    { 3 }
12448 \int_const:Nn \c__fp_prec_comma_int    { 2 }
12449 \int_const:Nn \c__fp_prec_tuple_int    { 1 }
12450 \int_const:Nn \c__fp_prec_end_int       { 0 }

```

(End definition for `\c__fp_prec_func_int` and others.)

25.1.1 Storing results

The main question in parsing expressions expandably is to decide where to put the intermediate results computed for various subexpressions.

One option is to store the values at the start of the expression, and carry them together as the first argument of each macro. However, we want to **f**-expand tokens one by one in the expression (as `\int_eval:n` does), and with this approach, expanding the next unread token forces us to jump with `\exp_after:wN` over every value computed earlier in the expression. With this approach, the run-time grows at least quadratically in the length of the expression, if not as its cube (inserting the `\exp_after:wN` is tricky and slow).

A second option is to place those values at the end of the expression. Then expanding the next unread token is straightforward, but this still hits a performance issue: for long expressions we would be reaching all the way to the end of the expression at every step of the calculation. The run-time is again quadratic.

A variation of the above attempts to place the intermediate results which appear when computing a parenthesized expression near the closing parenthesis. This still lets us expand tokens as we go, and avoids performance problems as long as there are enough parentheses. However, it would be better to avoid requiring the closing parenthesis to be present as soon as the corresponding opening parenthesis is read: the closing parenthesis may still be hidden in a macro yet to be expanded.

Hence, we need to go for some fine expansion control: the result is stored *before* the start!

Let us illustrate this idea in a simple model: adding positive integers which may be resulting from the expansion of macros, or may be values of registers. Assume that one number, say, 12345, has already been found, and that we want to parse the next number. The current status of the code may look as follows.

```
\exp_after:wN \add:ww \int_value:w 12345 \exp_after:wN ;
\exp:w \operand:w <stuff>
```

One step of expansion expands `\exp_after:wN`, which triggers the primitive `\int_value:w`, which reads the five digits we have already found, 12345. This integer is unfinished, causing the second `\exp_after:wN` to expand, and to trigger the construction `\exp:w`, which expands `\operand:w`, defined to read what follows and make a number out of it, then leave `\exp_end:`, the number, and a semicolon in the input stream. Once `\operand:w` is done expanding, we obtain essentially

```
\exp_after:wN \add:ww \int_value:w 12345 ;
\exp:w \exp_end: 333444 ;
```

where in fact `\exp_after:wN` has already been expanded, `\int_value:w` has already seen 12345, and `\exp:w` is still looking for a number. It finds `\exp_end:`, hence expands to nothing. Now, `\int_value:w` sees the `;`, which cannot be part of a number. The expansion stops, and we are left with

```
\add:ww 12345 ; 333444 ;
```

which can safely perform the addition by grabbing two arguments delimited by `;`.

If we were to continue parsing the expression, then the following number should also be cleaned up before the next use of a binary operation such as `\add:ww`. Just like `\int_value:w 12345 \exp_after:wN ;` expanded what follows once, we need `\add:ww` to do the calculation, and in the process to expand the following once. This is also true in our real application: all the functions of the form `_fp_..._o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we need to be careful not to waste this memory. All of our discussion above is nice but simplistic, as operations should not simply be performed in the order they appear.

25.1.2 Precedence and infix operators

The various operators we will encounter have different precedences, which influence the order of calculations: $1 + 2 \times 3 = 1 + (2 \times 3)$ because \times has a higher precedence than $+$. The true analog of our macro `\operand:w` must thus take care of that. When looking for an operand, it needs to perform calculations until reaching an operator which has lower precedence than the one which called `\operand:w`. This means that `\operand:w` must know what the previous binary operator is, or rather, its precedence: we thus rename it `\operand:Nw`. Let us describe as an example how we plan to do the calculation

$41-2^3*4+5$. More precisely we describe how to perform the first operation in this expression. Here, we abuse notations: the first argument of `\operand:Nw` should be an integer constant (`\c__fp_prec_plus_int, ...`) equal to the precedence of the given operator, not directly the operator itself.

- Clean up 41 and find `-`. We call `\operand:Nw -` to find the second operand.
- Clean up 2 and find `^`.
- Compare the precedences of `-` and `^`. Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw ^`.
- Clean up 3 and find `*`.
- Compare the precedences of `^` and `*`. Since the former is higher, `\operand:Nw ^` has found the second operand of the exponentiation, which is computed: $2^3 = 8$.
- We now have $41-8*4+5$, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 8?
- Compare the precedences of `-` and `*`. Since the latter is higher, we are not done with 8. Call `\operand:Nw *` to find the second operand of the multiplication.
- Clean up 4, and find `+`.
- Compare the precedences of `*` and `+`. Since the former is higher, `\operand:Nw *` has found the second operand of the multiplication, which is computed: $8*4 = 32$.
- We now have $41-32+5$, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 32?
- Compare the precedences of `-` and `+`. Since they are equal, `\operand:Nw -` has found the second operand for the subtraction, which is computed: $41 - 32 = 9$.
- We now have $9+5$.

The procedure above stops short of performing all computations, but adding a surrounding call to `\operand:Nw` with a very low precedence ensures that all computations are performed before `\operand:Nw` is done. Adding a trailing marker with the same very low precedence prevents the surrounding `\operand:Nw` from going beyond the marker.

The pattern above to find an operand for a given operator, is to find one number and the next operator, then compare precedences to know if the next computation should be done. If it should, then perform it after finding its second operand, and look at the next operator, then compare precedences to know if the next computation should be done. This continues until we find that the next computation should not be done. Then, we stop.

We are now ready to get a bit more technical and describe which of the `l3fp-parse` functions correspond to each step above.

First, `__fp_parse_operand:Nw` is the `\operand:Nw` function above, with small modifications due to expansion issues discussed later. We denote by $\langle precedence \rangle$ the argument of `__fp_parse_operand:Nw`, that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by `__fp_parse_one:Nw`. A first approximation of this function is that it reads one $\langle number \rangle$, performing no computation, and finds the following binary $\langle operator \rangle$. Then it expands to

```

    <number>
    \__fp_parse_infix_<operator>:N <precedence>

```

expanding the `infix` auxiliary before leaving the above in the input stream.

We now explain the `infix` auxiliaries. We need some flexibility in how we treat the case of equal precedences: most often, the first operation encountered should be performed, such as `1-2-3` being computed as `(1-2)-3`, but `2^3^4` should be evaluated as `2^(3^4)` instead. For this reason, and to support the equivalence between `**` and `^` more easily, each binary operator is converted to a control sequence `__fp_parse_infix_<operator>:N` when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the `<precedence>` (of the earlier operator) to the `infix` auxiliary for the following `<operator>`, to know whether to perform the computation of the `<operator>`. If it should not be performed, the `infix` auxiliary expands to

```

    @ \use_none:n \__fp_parse_infix_<operator>:N

```

and otherwise it calls `__fp_parse_operand:Nw` with the precedence of the `<operator>` to find its second operand `<number2 and the next <operator2, and expands to`

```

    @ \__fp_parse_apply_binary:NwNwN
      <operator> <number22

```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand `<number>` is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `__fp_parse_operand:Nw <precedence>` with some of the expansion control removed is

```

    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN <precedence>
    \exp:w \exp_end_continue_f:w
      \__fp_parse_one:Nw <precedence>

```

This expands `__fp_parse_one:Nw <precedence>` completely, which finds a number, wraps the next `<operator>` into an `infix` function, feeds this function the `<precedence>`, and expands it, yielding either

```

    \__fp_parse_continue:NwN <precedence>
    <number> @
    \use_none:n \__fp_parse_infix_<operator>:N

```

or

```

    \__fp_parse_continue:NwN <precedence>
    <number> @
    \__fp_parse_apply_binary:NwNwN
      <operator> <number22

```

The definition of `__fp_parse_continue:NwN` is then very simple:

```

    \cs_new:Npn \__fp_parse_continue:NwN #1#2@#3 { #3 #1 #2 @ }

```


In the first case, #3 is `\use_none:n`, yielding

```
\use_none:n <precedence> <number> @
\__fp_parse_infix_<operator>:N
```

then `<number> @ __fp_parse_infix_<operator>:N`. In the second case, #3 is `__fp_parse_apply_binary:NwNwN`, whose role is to compute `<number> <operator> <number>` and to prepare for the next comparison of precedences: first we get

```
\__fp_parse_apply_binary:NwNwN
  <precedence> <number> @
  <operator> <number>
@ \__fp_parse_infix_<operator>:N
```

then

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN <precedence>
\exp:w \exp_end_continue_f:w
\__fp_<operator>_o:ww <number> <number>
\exp:w \exp_end_continue_f:w
\__fp_parse_infix_<operator>:N <precedence>
```

where `__fp_<operator>_o:ww` computes `<number> <operator> <number>` and expands after the result, thus triggers the comparison of the precedence of the `<operator>` and the `<precedence>`, continuing the loop.

We have introduced the most important functions here, and the next few paragraphs we describe various subtleties.

25.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary `-`, `+`, `!`) and parentheses are taken care of by the same mechanism, and functions (`sin`, `exp`, etc.) as well. Finding the argument of the unary `-`, for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary `-`, this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function `__fp_parse_one:Nw` earlier. This function is responsible for reading in the input stream the first possible `<number>` and the next infix `<operator>`. If what follows `__fp_parse_one:Nw <precedence>` is a prefix operator, then we must find the operand of this prefix operator through a nested call to `__fp_parse_operand:Nw` with the appropriate precedence, then apply the operator to the operand found to yield the result of `__fp_parse_one:Nw`. So far, all is simple.

The unary operators `+`, `-`, `!` complicate things a little bit: `-3**2` should be $-(3^2) = -9$, and not $(-3)^2 = 9$. This would easily be done by giving `-` a lower precedence, equal to that of the infix `+` and `-`. Unfortunately, this fails in cases such as `3**-2*4`, yielding $3^{-2 \times 4}$ instead of the correct $3^{-2} \times 4$. A second attempt would be to call `__fp_parse_operand:Nw` with the `<precedence>` of the previous operator, but `0>-2+3` is

then parsed as `0>-(2+3)`: the addition is performed because it binds more tightly than the comparison which precedes `-`. The correct approach is for a unary `-` to perform operations whose precedence is greater than both that of the previous operation, and that of the unary `-` itself. The unary `-` is given a precedence higher than multiplication and division. This does not lead to any surprising result, since $-(x/y) = (-x)/y$ and similarly for multiplication, and it reduces the number of nested calls to `__fp_parse_operand:Nw`.

Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the `infix` functions discussed earlier, the `prefix` functions do perform tests on the previous *precedence* to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

25.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a “number”? A number is typically given in the form $\langle \textit{significand} \rangle \mathbf{e} \langle \textit{exponent} \rangle$, where the $\langle \textit{significand} \rangle$ is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “ $\mathbf{e} \langle \textit{exponent} \rangle$ ” is optional and is composed of an exponent mark `e` followed by a possibly empty string of signs `+` or `-` and a non-empty string of decimal digits. The $\langle \textit{significand} \rangle$ can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the $\langle \textit{exponent} \rangle$ can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as `nan`, `inf` or `pi`. We may add more types in the future.

When `__fp_parse_one:Nw` is looking for a “number”, here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s__fp`, in which case our job is done, as what follows is an internal floating point number, or `\s__fp_mark`, in which case the expression has come to an early end, as we are still looking for a number here, or something else, in which case we consider the control sequence to be a bad variable resulting from `c`-expansion.
- If the next token is a control sequence with a different meaning, we assume that it is a register, unpack it with `\tex_the:D`, and use its value (in `pt` for dimensions and skips, `mu` for muskips) as the $\langle \textit{significand} \rangle$ of a number: we look for an exponent.
- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.
- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as `asin`, a constant such as `pi` or be unknown. In the first case, we call `__fp_parse_operand:Nw` to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value `nan` for unknown words.
- If the next token is anything else, we check whether it is a known prefix operator, in which case `__fp_parse_operand:Nw` finds its operand. If it is not known, then either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, `__fp_parse_one:Nw` also finds an infix operator. This goes as follows.

- If the next token is a control sequence, it could be the special marker `\s__fp_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_zero`, with an implied multiplication.
- If the next token is a letter, it is also a case of juxtaposition, as letters cannot be proper infix operators.
- Otherwise (including in the case of digits), if the token is a known infix operator, the appropriate `__fp_infix_⟨operator⟩:N` function is built, and if it does not exist, we complain. In particular, the juxtaposition `\c_zero 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

```
\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
  not a digit
\fi:
```

To exclude 0, replace 9 by 10. The use of `\token_to_str:N` ensures that a digit with any catcode is detected. To test if a character token is a letter, we need to work with its character code, testing if `'#1` lies in [65, 90] (uppercase letters) or [97, 112] (lowercase letters)

```
\if_int_compare:w \__fp_int_eval:w
  ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26 = 3 \exp_stop_f:
  is a letter
\else:
  not a letter
\fi:
```

At all steps, we try to accept all category codes: when `#1` is kept to be used later, it is almost always converted to category code other through `\token_to_str:N`. More precisely, catcodes {3, 6, 7, 8, 11, 12} should work without trouble, but not {1, 2, 4, 10, 13}, and of course {0, 5, 9} cannot become tokens.

Floating point expressions should behave as much as possible like ε -TeX-based integer expressions and dimension expressions. In particular, `f`-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces. One advantage of expanding at every step is that restricted expandable functions can then be used in floating point expressions just as they can be in other kinds of expressions. Problematically, spaces stop `f`-expansion: for instance, the macro `\X` below would not be expanded if we simply performed `f`-expansion.

```
\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }
```

Of course, spaces typically do not appear in a code setting, but may very easily come in document-level input, from which some expressions may come. To avoid this problem, at every step, we do essentially what `\use:f` would do: take an argument, put it back

in the input stream, then **f**-expand it. This is not a complete solution, since a macro's expansion could contain leading spaces which would stop the **f**-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers are correctly expanded to the underlying `\s__fp ...` structure. The **f**-expansion is performed by `__fp_parse_expand:w`.

25.2 Main auxiliary functions

`__fp_parse_operand:Nw` `\exp:w __fp_parse_operand:Nw <precedence> __fp_parse_expand:w`
 Reads the "...", performing every computation with a precedence higher than `<precedence>`, then expands to

`<result> @ __fp_parse_infix_<operation>:N ...`

where the `<operation>` is the first operation with a lower precedence, possibly **end**, and the "..." start just after the `<operation>`.

(End definition for `__fp_parse_operand:Nw`.)

`__fp_parse_infix_+:N` `__fp_parse_infix_+:N <precedence> ...`
 If `+` has a precedence higher than the `<precedence>`, cleans up a second `<operand>` and finds the `<operation2 which follows, and expands to`

`@ __fp_parse_apply_binary:NwNwN + <operand> @ __fp_parse_infix_<operation2>:N ...`

Otherwise expands to

`@ \use_none:n __fp_parse_infix_+:N ...`

A similar function exists for each infix operator.

(End definition for `__fp_parse_infix_+:N`.)

`__fp_parse_one:Nw` `__fp_parse_one:Nw <precedence> ...`
 Cleans up one or two operands depending on how the precedence of the next operation compares to the `<precedence>`. If the following `<operation>` has a precedence higher than `<precedence>`, expands to

`<operand1> @ __fp_parse_apply_binary:NwNwN <operation> <operand2> @ __fp_parse_infix_<operation2>:N ...`

and otherwise expands to

`<operand> @ \use_none:n __fp_parse_infix_<operation>:N ...`

(End definition for `__fp_parse_one:Nw`.)

25.3 Helpers

`__fp_parse_expand:w` `\exp:w __fp_parse_expand:w <tokens>`

This function must always come within a `\exp:w` expansion. The `<tokens>` should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

12451 `\cs_new:Npn __fp_parse_expand:w #1 { \exp_end_continue_f:w #1 }`

(End definition for `__fp_parse_expand:w`.)

`__fp_parse_return_semicolon:w` This very odd function swaps its position with the following `\fi:` and removes `__fp_parse_expand:w` normally responsible for expansion. That turns out to be useful.

12452 `\cs_new:Npn __fp_parse_return_semicolon:w`
12453 `#1 \fi: __fp_parse_expand:w { \fi: ; #1 }`

(End definition for `__fp_parse_return_semicolon:w`.)

`__fp_parse_digits_vii:N` These functions must be called within an `\int_value:w` or `__fp_int_eval:w` construction. The first token which follows must be `f`-expanded prior to calling those functions. The functions read tokens one by one, and output digits into the input stream, until meeting a non-digit, or up to a number of digits equal to their index. The full expansion is

`__fp_parse_digits_vii:N` `<digits> ; <filling 0> ; <length>`
`__fp_parse_digits_vi:N`
`__fp_parse_digits_v:N`
`__fp_parse_digits_iv:N`
`__fp_parse_digits_iii:N`
`__fp_parse_digits_ii:N`
`__fp_parse_digits_i:N`
`__fp_parse_digits_:N`

where `<filling 0>` is a string of zeros such that `<digits> <filling 0>` has the length given by the index of the function, and `<length>` is the number of zeros in the `<filling 0>` string. Each function puts a digit into the input stream and calls the next function, until we find a non-digit. We are careful to pass the tested tokens through `\token_to_str:N` to normalize their category code.

12454 `\cs_set_protected:Npn __fp_tmp:w #1 #2 #3`
12455 `{`
12456 `\cs_new:cpn { __fp_parse_digits_ #1 :N } ##1`
12457 `{`
12458 `\if_int_compare:w 9 < 1 \token_to_str:N ##1 \exp_stop_f:`
12459 `\token_to_str:N ##1 \exp_after:wN #2 \exp:w`
12460 `\else:`
12461 `__fp_parse_return_semicolon:w #3 ##1`
12462 `\fi:`
12463 `__fp_parse_expand:w`
12464 `}`
12465 `}`
12466 `__fp_tmp:w {vii} __fp_parse_digits_vi:N { 0000000 ; 7 }`
12467 `__fp_tmp:w {vi} __fp_parse_digits_v:N { 000000 ; 6 }`
12468 `__fp_tmp:w {v} __fp_parse_digits_iv:N { 00000 ; 5 }`
12469 `__fp_tmp:w {iv} __fp_parse_digits_iii:N { 0000 ; 4 }`
12470 `__fp_tmp:w {iii} __fp_parse_digits_ii:N { 000 ; 3 }`
12471 `__fp_tmp:w {ii} __fp_parse_digits_i:N { 00 ; 2 }`
12472 `__fp_tmp:w {i} __fp_parse_digits_:N { 0 ; 1 }`
12473 `\cs_new:Npn __fp_parse_digits_:N { ; ; 0 }`

(End definition for `__fp_parse_digits_vii:N` and others.)

25.4 Parsing one number

`__fp_parse_one:Nw` This function finds one number, and packs the symbol which follows in an `__fp_parse_infix_...` csname. #1 is the previous *precedence*, and #2 the first token of the operand. We distinguish four cases: #2 is equal to `\scan_stop:` in meaning, #2 is a different control sequence, #2 is a digit, and #2 is something else (this last case is split further later). Despite the earlier f-expansion, #2 may still be expandable if it was protected by `\exp_not:N`, as may happen with the L^AT_EX 2_ε command `\protect`. Using a well placed `\reverse_if:N`, this case is sent to `__fp_parse_one_fp:NN` which deals with it robustly.

```

12474 \cs_new:Npn \__fp_parse_one:Nw #1 #2
12475 {
12476   \if_catcode:w \scan_stop: \exp_not:N #2
12477     \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
12478       \exp_after:wN \reverse_if:N
12479     \fi:
12480     \if_meaning:w \scan_stop: #2
12481       \exp_after:wN \exp_after:wN
12482       \exp_after:wN \__fp_parse_one_fp:NN
12483     \else:
12484       \exp_after:wN \exp_after:wN
12485       \exp_after:wN \__fp_parse_one_register:NN
12486     \fi:
12487   \else:
12488     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
12489       \exp_after:wN \exp_after:wN
12490       \exp_after:wN \__fp_parse_one_digit:NN
12491     \else:
12492       \exp_after:wN \exp_after:wN
12493       \exp_after:wN \__fp_parse_one_other:NN
12494     \fi:
12495   \fi:
12496   #1 #2
12497 }
```

(End definition for `__fp_parse_one:Nw`.)

`__fp_parse_one_fp:NN` This function receives a *precedence* and a control sequence equal to `\scan_stop:` in meaning. There are three cases.

`__fp_exp_after_mark_f:nw`

`__fp_exp_after_?_f:nw`

- `\s__fp` starts a floating point number, and we call `__fp_exp_after_f:nw`, which f-expands after the floating point.
- `\s__fp_mark` is a premature end, we call `__fp_exp_after_mark_f:nw`, which triggers an fp-early-end error.
- For a control sequence not containing `\s__fp`, we call `__fp_exp_after_?_f:nw`, causing a bad-variable error.

This scheme is extensible: additional types can be added by starting the variables with a scan mark of the form `\s__fp_⟨type⟩` and defining `__fp_exp_after_⟨type⟩_f:nw`. In all cases, we make sure that the second argument of `__fp_parse_infix:NN` is correctly expanded. A special case only enabled in L^AT_EX 2_ε is that if `\protect` is encountered then

the error message mentions the control sequence which follows it rather than `\protect` itself. The test for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}_{2\epsilon}$ uses `\@unexpandable@protect` rather than `\protect` because `\protect` is often `\scan_stop:` hence “does not exist”.

```

12498 \cs_new:Npn \__fp_parse_one_fp:NN #1
12499 {
12500   \__fp_exp_after_any_f:nw
12501   {
12502     \exp_after:wN \__fp_parse_infix:NN
12503     \exp_after:wN #1 \exp:w \__fp_parse_expand:w
12504   }
12505 }
12506 \cs_new:Npn \__fp_exp_after_mark_f:nw #1
12507 {
12508   \int_case:nnF { \exp_after:wN \use_i:nnn \use_none:nnn #1 }
12509   {
12510     \c__fp_prec_comma_int { }
12511     \c__fp_prec_tuple_int { }
12512     \c__fp_prec_end_int
12513     {
12514       \exp_after:wN \c__fp_empty_tuple_fp
12515       \exp:w \exp_end_continue_f:w
12516     }
12517   }
12518   {
12519     \__kernel_msg_expandable_error:nn { kernel } { fp-early-end }
12520     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
12521   }
12522   #1
12523 }
12524 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
12525 {
12526   \__kernel_msg_expandable_error:nnn { kernel } { bad-variable }
12527   {#2}
12528   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
12529 }
12530 \*package>
12531 \cs_set_protected:Npn \__fp_tmp:w #1
12532 {
12533   \cs_if_exist:NT #1
12534   {
12535     \cs_gset:cpn { __fp_exp_after_?_f:nw } ##1##2
12536     {
12537       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w ##1
12538       \str_if_eq:nnTF {##2} { \protect }
12539       {
12540         \cs_if_eq:NNTF ##2 #1 { \use_i:nn } { \use:n }
12541         {
12542           \__kernel_msg_expandable_error:nnn { kernel }
12543           { fp-robust-cmd }
12544         }
12545       }
12546       {
12547         \__kernel_msg_expandable_error:nnn { kernel }
12548         { bad-variable } {##2}

```

```

12549         }
12550     }
12551 }
12552 }
12553 \exp_args:Nc \__fp_tmp:w { @unexpandable@protect }
12554 \</package>

```

(End definition for `__fp_parse_one_fp:NN`, `__fp_exp_after_mark_f:nw`, and `__fp_exp_after_?-f:nw`.)

```

\__fp_parse_one_register:NN
  \__fp_parse_one_register_aux:Nw
  \__fp_parse_one_register_auxii:wwwNw
  \__fp_parse_one_register_int:www
  \__fp_parse_one_register_mu:www
  \__fp_parse_one_register_dim:ww

```

This is called whenever #2 is a control sequence other than `\scan_stop:` in meaning. We special-case `\wd`, `\ht`, `\dp` (see later) and otherwise assume that it is a register, but carefully unpack it with `\tex_the:D` within braces. First, we find the exponent following #2. Then we unpack #2 with `\tex_the:D`, and the `auxii` auxiliary distinguishes integer registers from dimensions/skips from muskips, according to the presence of a period and/or of `pt`. For integers, simply convert $\langle value \rangle e \langle exponent \rangle$ to a floating point number with `__fp_parse:n` (this is somewhat wasteful). For other registers, the decimal rounding provided by TeX does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with `\int_value:w \dim_to_decimal_in_sp:n { \langle decimal value \rangle pt }`, and use an auxiliary of `\dim_to_fp:n`, which performs the multiplication by 2^{-16} , correctly rounded.

```

12555 \cs_new:Npn \__fp_parse_one_register:NN #1#2
12556 {
12557   \exp_after:wN \__fp_parse_infix_after_operand:NwN
12558   \exp_after:wN #1
12559   \exp:w \exp_end_continue_f:w
12560   \__fp_parse_one_register_special:N #2
12561   \exp_after:wN \__fp_parse_one_register_aux:Nw
12562   \exp_after:wN #2
12563   \int_value:w
12564   \exp_after:wN \__fp_parse_exponent:N
12565   \exp:w \__fp_parse_expand:w
12566 }
12567 \cs_new:Npx \__fp_parse_one_register_aux:Nw #1
12568 {
12569   \exp_not:n
12570   {
12571     \exp_after:wN \use:nn
12572     \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
12573   }
12574   \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
12575   ; \exp_not:N \__fp_parse_one_register_dim:ww
12576   \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_mu:www
12577   . \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_int:www
12578   \exp_not:N \q_stop
12579 }
12580 \exp_args:Nno \use:nn
12581 { \cs_new:Npn \__fp_parse_one_register_auxii:wwwNw #1 . #2 }
12582 { \tl_to_str:n { pt } #3 ; #4#5 \q_stop }
12583 { #4 #1.#2; }
12584 \exp_args:Nno \use:nn
12585 { \cs_new:Npn \__fp_parse_one_register_mu:www #1 }
12586 { \tl_to_str:n { mu } ; #2 ; }
12587 { \__fp_parse_one_register_dim:ww #1 ; }

```



```

12588 \cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
12589 { \__fp_parse:n { #1 e #3 } }
12590 \cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;
12591 {
12592   \exp_after:wN \__fp_from_dim_test:ww
12593   \int_value:w #2 \exp_after:wN ,
12594   \int_value:w \dim_to_decimal_in_sp:n { #1 pt } ;
12595 }

```

(End definition for __fp_parse_one_register:NN and others.)

```

\__fp_parse_one_register_special:N
\__fp_parse_one_register_math:NNw
  \__fp_parse_one_register_wd:w
  \__fp_parse_one_register_wd:Nw

```

The \wd, \dp, \ht primitives expect an integer argument. We abuse the exponent parser to find the integer argument: simply include the exponent marker e. Once that “exponent” is found, use \tex_the:D to find the box dimension and then copy what we did for dimensions.

```

12596 \cs_new:Npn \__fp_parse_one_register_special:N #1
12597 {
12598   \if_meaning:w \box_wd:N #1 \__fp_parse_one_register_wd:w \fi:
12599   \if_meaning:w \box_ht:N #1 \__fp_parse_one_register_wd:w \fi:
12600   \if_meaning:w \box_dp:N #1 \__fp_parse_one_register_wd:w \fi:
12601   \if_meaning:w \infty #1
12602     \__fp_parse_one_register_math:NNw \infty #1
12603   \fi:
12604   \if_meaning:w \pi #1
12605     \__fp_parse_one_register_math:NNw \pi #1
12606   \fi:
12607 }
12608 \cs_new:Npn \__fp_parse_one_register_math:NNw
12609   #1#2#3#4 \__fp_parse_expand:w
12610 {
12611   #3
12612   \str_if_eq:nnTF {#1} {#2}
12613   {
12614     \__kernel_msg_expandable_error:nnn
12615     { kernel } { fp-infty-pi } {#1}
12616     \c_nan_fp
12617   }
12618   { #4 \__fp_parse_expand:w }
12619 }
12620 \cs_new:Npn \__fp_parse_one_register_wd:w
12621   #1#2 \exp_after:wN #3#4 \__fp_parse_expand:w
12622 {
12623   #1
12624   \exp_after:wN \__fp_parse_one_register_wd:Nw
12625   #4 \__fp_parse_expand:w e
12626 }
12627 \cs_new:Npn \__fp_parse_one_register_wd:Nw #1#2 ;
12628 {
12629   \exp_after:wN \__fp_from_dim_test:ww
12630   \exp_after:wN 0 \exp_after:wN ,
12631   \int_value:w \dim_to_decimal_in_sp:n { #1 #2 } ;
12632 }

```

(End definition for __fp_parse_one_register_special:N and others.)

`__fp_parse_one_digit:NN` A digit marks the beginning of an explicit floating point number. Once the number is found, we catch the case of overflow and underflow with `__fp_sanitize:wN`, then `__fp_parse_infix_after_operand:NwN` expands `__fp_parse_infix:NN` after the number we find, to wrap the following infix operator as required. Finding the number itself begins by removing leading zeros: further steps are described later.

```

12633 \cs_new:Npn \__fp_parse_one_digit:NN #1
12634 {
12635     \exp_after:wN \__fp_parse_infix_after_operand:NwN
12636     \exp_after:wN #1
12637     \exp:w \exp_end_continue_f:w
12638     \exp_after:wN \__fp_sanitize:wN
12639     \int_value:w \__fp_int_eval:w 0 \__fp_parse_trim_zeros:N
12640 }

```

(End definition for `__fp_parse_one_digit:NN`.)

`__fp_parse_one_other:NN` For this function, #2 is a character token which is not a digit. If it is an ASCII letter, `__fp_parse_letters:N` beyond this one and give the result to `__fp_parse_word:Nw`. Otherwise, the character is assumed to be a prefix operator, and we build `__fp_parse_prefix_{operator}:Nw`.

```

12641 \cs_new:Npn \__fp_parse_one_other:NN #1 #2
12642 {
12643     \if_int_compare:w
12644         \__fp_int_eval:w
12645         ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
12646         = 3 \exp_stop_f:
12647         \exp_after:wN \__fp_parse_word:Nw
12648         \exp_after:wN #1
12649         \exp_after:wN #2
12650         \exp:w \exp_after:wN \__fp_parse_letters:N
12651         \exp:w
12652     \else:
12653         \exp_after:wN \__fp_parse_prefix:NNN
12654         \exp_after:wN #1
12655         \exp_after:wN #2
12656         \cs:w
12657         __fp_parse_prefix_ \token_to_str:N #2 :Nw
12658         \exp_after:wN
12659         \cs_end:
12660         \exp:w
12661     \fi:
12662     \__fp_parse_expand:w
12663 }

```

(End definition for `__fp_parse_one_other:NN`.)

`__fp_parse_word:Nw` Finding letters is a simple recursion. Once `__fp_parse_letters:N` has done its job, `__fp_parse_letters:N` we try to build a control sequence from the word #2. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield `\c_nan_fp`, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not. The standard requires “inf” and “infinity” and “nan” to be recognized regardless of case, but we probably don’t want to allow every l3fp word to

have an arbitrary mixture of lower and upper case, so we test and use a differently-named control sequence.

```

12664 \cs_new:Npn \__fp_parse_word:Nw #1#2;
12665 {
12666   \cs_if_exist_use:cF { __fp_parse_word_#2:N }
12667   {
12668     \cs_if_exist_use:cF
12669     { __fp_parse_caseless_ \str_fold_case:n {#2} :N }
12670     {
12671       \__kernel_msg_expandable_error:nnn
12672       { kernel } { unknown-fp-word } {#2}
12673       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
12674       \__fp_parse_infix:NN
12675     }
12676   }
12677   #1
12678 }
12679 \cs_new:Npn \__fp_parse_letters:N #1
12680 {
12681   \exp_end_continue_f:w
12682   \if_int_compare:w
12683   \if_catcode:w \scan_stop: \exp_not:N #1
12684   0
12685   \else:
12686     \__fp_int_eval:w
12687     ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26
12688     \fi:
12689     = 3 \exp_stop_f:
12690     \exp_after:wN #1
12691     \exp:w \exp_after:wN \__fp_parse_letters:N
12692     \exp:w
12693   \else:
12694     \__fp_parse_return_semicolon:w #1
12695     \fi:
12696     \__fp_parse_expand:w
12697   }

```

(End definition for __fp_parse_word:Nw and __fp_parse_letters:N.)

__fp_parse_prefix:NNN
 __fp_parse_prefix_unknown:NNN

For this function, #1 is the previous *precedence*, #2 is the operator just seen, and #3 is a control sequence which implements the operator if it is a known operator. If this control sequence is \scan_stop:, then the operator is in fact unknown. Either the expression is missing a number there (if the operator is valid as an infix operator), and we put `nan`, wrapping the infix operator in a csname as appropriate, or the character is simply invalid in floating point expressions, and we continue looking for a number, starting again from __fp_parse_one:Nw.

```

12698 \cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
12699 {
12700   \if_meaning:w \scan_stop: #3
12701   \exp_after:wN \__fp_parse_prefix_unknown:NNN
12702   \exp_after:wN #2
12703   \fi:
12704   #3 #1

```

```

12705     }
12706 \cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
12707 {
12708     \cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
12709     {
12710         \__kernel_msg_expandable_error:nnn
12711         { kernel } { fp-missing-number } {#1}
12712         \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
12713         \__fp_parse_infix:NN #3 #1
12714     }
12715     {
12716         \__kernel_msg_expandable_error:nnn
12717         { kernel } { fp-unknown-symbol } {#1}
12718         \__fp_parse_one:Nw #3
12719     }
12720 }

```

(End definition for __fp_parse_prefix:NNN and __fp_parse_prefix_unknown:NNN.)

25.4.1 Numbers: trimming leading zeros

Numbers are parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand ≥ 1 with the set of functions __fp_parse_large...; if it is a period, the significand is < 1 ; and otherwise it is zero. In the second case, trim additional zeros after the period, counting them for an exponent shift $\langle exp_1 \rangle < 0$, then read the significand with the set of functions __fp_parse_small... Once the significand is read, read the exponent if e is present.

__fp_parse_trim_zeros:N This function expects an already expanded token. It removes any leading zero, then distinguishes three cases: if the first non-zero token is a digit, then call __fp_parse_large:N (the significand is ≥ 1); if it is ., then continue trimming zeros with __fp_parse_strim_zeros:N; otherwise, our number is exactly zero, and we call __fp_parse_zero: to take care of that case.

```

12721 \cs_new:Npn \__fp_parse_trim_zeros:N #1
12722 {
12723     \if:w 0 \exp_not:N #1
12724         \exp_after:wN \__fp_parse_trim_zeros:N
12725         \exp:w
12726     \else:
12727         \if:w . \exp_not:N #1
12728             \exp_after:wN \__fp_parse_strim_zeros:N
12729             \exp:w
12730         \else:
12731             \__fp_parse_trim_end:w #1
12732         \fi:
12733     \fi:
12734     \__fp_parse_expand:w
12735 }
12736 \cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
12737 {
12738     \fi:
12739     \fi:
12740     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:

```

```

12741     \exp_after:wN \__fp_parse_large:N
12742   \else:
12743     \exp_after:wN \__fp_parse_zero:
12744   \fi:
12745   #1
12746 }

```

(End definition for __fp_parse_trim_zeros:N and __fp_parse_trim_end:w.)

__fp_parse_strim_zeros:N If we have removed all digits until a period (or if the body started with a period), then enter the “small_trim” loop which outputs −1 for each removed 0. Those −1 are added to an integer expression waiting for the exponent. If the first non-zero token is a digit, call __fp_parse_small:N (our significand is smaller than 1), and otherwise, the number is an exact zero. The name `strim` stands for “small trim”.

```

12747 \cs_new:Npn \__fp_parse_strim_zeros:N #1
12748 {
12749   \if:w 0 \exp_not:N #1
12750     - 1
12751     \exp_after:wN \__fp_parse_strim_zeros:N \exp:w
12752   \else:
12753     \__fp_parse_strim_end:w #1
12754   \fi:
12755   \__fp_parse_expand:w
12756 }
12757 \cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
12758 {
12759   \fi:
12760   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
12761     \exp_after:wN \__fp_parse_small:N
12762   \else:
12763     \exp_after:wN \__fp_parse_zero:
12764   \fi:
12765   #1
12766 }

```

(End definition for __fp_parse_strim_zeros:N and __fp_parse_strim_end:w.)

__fp_parse_zero: After reading a significand of 0, find any exponent, then put a sign of 1 for __fp-sanitize:wN, which removes everything and leaves an exact zero.

```

12767 \cs_new:Npn \__fp_parse_zero:
12768 {
12769   \exp_after:wN ; \exp_after:wN 1
12770   \int_value:w \__fp_parse_exponent:N
12771 }

```

(End definition for __fp_parse_zero:.)

25.4.2 Number: small significand

__fp_parse_small:N This function is called after we have passed the decimal separator and removed all leading zeros from the significand. It is followed by a non-zero digit (with any catcode). The goal is to read up to 16 digits. But we can’t do that all at once, because \int_value:w (which allows us to collect digits and continue expanding) can only go up to 9 digits. Hence we grab digits in two steps of 8 digits. Since #1 is a digit, read seven more digits

using `__fp_parse_digits_vii:N`. The `small_leading` auxiliary leaves those digits in the `\int_value:w`, and grabs some more, or stops if there are no more digits. Then the `pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```

12772 \cs_new:Npn \__fp_parse_small:N #1
12773 {
12774   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
12775   \int_value:w \__fp_int_eval:w 1 \token_to_str:N #1
12776   \exp_after:wN \__fp_parse_small_leading:wwNN
12777   \int_value:w 1
12778   \exp_after:wN \__fp_parse_digits_vii:N
12779   \exp:w \__fp_parse_expand:w
12780 }

```

(End definition for `__fp_parse_small:N`.)

`__fp_parse_small_leading:wwNN` `__fp_parse_small_leading:wwNN 1 <digits> ; <zeros> ; <number of zeros>`

We leave `<digits>` `<zeros>` in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack 8 more digits, with an exponent shift of zero (this shift is used in the case of a large significand). If #4 is a digit, leave it behind for the packing function, and read 6 more digits to reach a total of 15 digits: further digits are involved in the rounding. Otherwise put 8 zeros in to complete the significand, then look for an exponent.

```

12781 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
12782 {
12783   #1 #2
12784   \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
12785   \exp_after:wN 0
12786   \int_value:w \__fp_int_eval:w 1
12787   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
12788     \token_to_str:N #4
12789     \exp_after:wN \__fp_parse_small_trailing:wwNN
12790     \int_value:w 1
12791     \exp_after:wN \__fp_parse_digits_vi:N
12792     \exp:w
12793   \else:
12794     0000 0000 \__fp_parse_exponent:Nw #4
12795   \fi:
12796   \__fp_parse_expand:w
12797 }

```

(End definition for `__fp_parse_small_leading:wwNN`.)

`__fp_parse_small_trailing:wwNN` `__fp_parse_small_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>`
 `<next token>`

Leave digits 10 to 15 (arguments #1 and #2) in the input stream. If the `<next token>` is a digit, it is the 16th digit, we keep it, then the `small_round` auxiliary considers this digit and all further digits to perform the rounding: the function expands to nothing, to +0 or to +1. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```

12798 \cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
12799 {
12800   #1 #2
12801   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:

```

```

12802     \token_to_str:N #4
12803     \exp_after:wN \__fp_parse_small_round:NN
12804     \exp_after:wN #4
12805     \exp:w
12806     \else:
12807       0 \__fp_parse_exponent:Nw #4
12808     \fi:
12809     \__fp_parse_expand:w
12810   }

```

(End definition for `__fp_parse_small_trailing:wwNN`.)

```

\__fp_parse_pack_trailing:NNNNNNww
\__fp_parse_pack_leading:NNNNNNww
\__fp_parse_pack_carry:w

```

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The `trailing` function has an exponent shift as its first argument, which we add to the exponent found in the `e...` syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (+1 in the code below). If the leading digits propagate this carry all the way up, the function `__fp_parse_pack_carry:w` increments the exponent, and changes the significand from 0000... to 1000...: this is simple because such a carry can only occur to give rise to a power of 10.

```

12811 \cs_new:Npn \__fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
12812 {
12813   \if_meaning:w 2 #2 + 1 \fi:
12814   ; #8 + #1 ; {#3#4#5#6} {#7};
12815 }
12816 \cs_new:Npn \__fp_parse_pack_leading:NNNNNNww #1 #2#3#4#5 #6; #7;
12817 {
12818   + #7
12819   \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
12820   ; 0 {#2#3#4#5} {#6}
12821 }
12822 \cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
12823 { \fi: + 1 ; 0 {1000} }

```

(End definition for `__fp_parse_pack_trailing:NNNNNNww`, `__fp_parse_pack_leading:NNNNNNww`, and `__fp_parse_pack_carry:w`.)

25.4.3 Number: large significand

Parsing a significand larger than 1 is a little bit more difficult than parsing small significands. We need to count the number of digits before the decimal separator, and add that to the final exponent. We also need to test for the presence of a dot each time we run out of digits, and branch to the appropriate `parse_small` function in those cases.

```

\__fp_parse_large:N

```

This function is followed by the first non-zero digit of a “large” significand (≥ 1). It is called within an integer expression for the exponent. Grab up to 7 more digits, for a total of 8 digits.

```

12824 \cs_new:Npn \__fp_parse_large:N #1
12825 {
12826   \exp_after:wN \__fp_parse_large_leading:wwNN
12827   \int_value:w 1 \token_to_str:N #1

```

```

12828     \exp_after:wN \__fp_parse_digits_vii:N
12829     \exp:w \__fp_parse_expand:w
12830 }

```

(End definition for __fp_parse_large:N.)

```

\__fp_parse_large_leading:wwNN    \__fp_parse_large_leading:wwNN 1 <digits> ; <zeros> ; <number of zeros>
                                   <next token>

```

We shift the exponent by the number of digits in #1, namely the target number, 8, minus the *<number of zeros>* (number of digits missing). Then prepare to pack the 8 first digits. If the *<next token>* is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the `small` functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the *<zeros>* to complete the 8 first digits, insert 8 more, and look for an exponent.

```

12831 \cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
12832 {
12833   + \c__fp_half_prec_int - #3
12834   \exp_after:wN \__fp_parse_pack_leading:NNNNNww
12835   \int_value:w \__fp_int_eval:w 1 #1
12836   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
12837     \exp_after:wN \__fp_parse_large_trailing:wwNN
12838     \int_value:w 1 \token_to_str:N #4
12839     \exp_after:wN \__fp_parse_digits_vi:N
12840     \exp:w
12841   \else:
12842     \if:w . \exp_not:N #4
12843       \exp_after:wN \__fp_parse_small_leading:wwNN
12844       \int_value:w 1
12845       \cs:w
12846         __fp_parse_digits_
12847         \__fp_int_to_roman:w #3
12848         :N \exp_after:wN
12849       \cs_end:
12850       \exp:w
12851     \else:
12852       #2
12853       \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
12854       \exp_after:wN 0
12855       \int_value:w 1 0000 0000
12856       \__fp_parse_exponent:Nw #4
12857     \fi:
12858   \fi:
12859   \__fp_parse_expand:w
12860 }

```

(End definition for __fp_parse_large_leading:wwNN.)

```

\__fp_parse_large_trailing:wwNN    \__fp_parse_large_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>
                                   <next token>

```

We have just read 15 digits. If the *<next token>* is a digit, then the exponent shift caused by this block of 8 digits is 8, first argument to the `pack_trailing` function. We keep the *<digits>* and this 16-th digit, and find how this should be rounded using `__fp_parse_large_round:NN`. Otherwise, the exponent shift is the number of *<digits>*,

7 minus the *⟨number of zeros⟩*, and we test for a decimal point. This case happens in 123451234512345.67 with exactly 15 digits before the decimal separator. Then branch to the appropriate `small` auxiliary, grabbing a few more digits to complement the digits we already grabbed. Finally, if this is truly the end of the significand, look for an exponent after using the *⟨zeros⟩* and providing a 16-th digit of 0.

```

12861 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
12862 {
12863   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
12864     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
12865     \exp_after:wN \c__fp_half_prec_int
12866     \int_value:w \__fp_int_eval:w 1 #1 \token_to_str:N #4
12867     \exp_after:wN \__fp_parse_large_round:NN
12868     \exp_after:wN #4
12869     \exp:w
12870   \else:
12871     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
12872     \int_value:w \__fp_int_eval:w 7 - #3 \exp_stop_f:
12873     \int_value:w \__fp_int_eval:w 1 #1
12874     \if:w . \exp_not:N #4
12875       \exp_after:wN \__fp_parse_small_trailing:wwNN
12876       \int_value:w 1
12877       \cs:w
12878         __fp_parse_digits_
12879         \__fp_int_to_roman:w #3
12880         :N \exp_after:wN
12881       \cs_end:
12882       \exp:w
12883     \else:
12884       #2 0 \__fp_parse_exponent:Nw #4
12885     \fi:
12886   \fi:
12887   \__fp_parse_expand:w
12888 }

```

(End definition for `__fp_parse_large_trailing:wwNN`.)

25.4.4 Number: beyond 16 digits, rounding

`__fp_parse_round_loop:N` This loop is called when rounding a number (whether the mantissa is small or large).
`__fp_parse_round_up:N` It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by ;0, otherwise by ;1. This is done by switching the loop to `round_up` at the first non-zero digit, thus we avoid to test whether digits are 0 or not once we see a first non-zero digit.

```

12889 \cs_new:Npn \__fp_parse_round_loop:N #1
12890 {
12891   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
12892     + 1
12893     \if:w 0 \token_to_str:N #1
12894       \exp_after:wN \__fp_parse_round_loop:N
12895       \exp:w
12896     \else:
12897       \exp_after:wN \__fp_parse_round_up:N

```

```

12898         \exp:w
12899         \fi:
12900     \else:
12901         \__fp_parse_return_semicolon:w 0 #1
12902     \fi:
12903     \__fp_parse_expand:w
12904 }
12905 \cs_new:Npn \__fp_parse_round_up:N #1
12906 {
12907     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
12908         + 1
12909         \exp_after:wN \__fp_parse_round_up:N
12910         \exp:w
12911     \else:
12912         \__fp_parse_return_semicolon:w 1 #1
12913     \fi:
12914     \__fp_parse_expand:w
12915 }

```

(End definition for __fp_parse_round_loop:N and __fp_parse_round_up:N.)

__fp_parse_round_after:wN After the loop __fp_parse_round_loop:N, this function fetches an exponent with __fp_parse_exponent:N, and combines it with the number of digits counted by __fp_parse_round_loop:N. At the same time, the result 0 or 1 is added to the surrounding integer expression.

```

12916 \cs_new:Npn \__fp_parse_round_after:wN #1; #2
12917 {
12918     + #2 \exp_after:wN ;
12919     \int_value:w \__fp_int_eval:w #1 + \__fp_parse_exponent:N
12920 }

```

(End definition for __fp_parse_round_after:wN.)

__fp_parse_small_round:NN Here, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If #2 is not a digit, then fetch an exponent and expand to ;*<exponent>* only. Otherwise, we expand to +0 or +1, then ;*<exponent>*. To decide which, call __fp_round_s:NNNw to know whether to round up, giving it as arguments a sign 0 (all explicit numbers are positive), the digit #1 to round, the first following digit #2, and either +0 or +1 depending on whether the following digits are all zero or not. This last argument is obtained by __fp_parse_round_loop:N, whose number of digits we discard by multiplying it by 0. The exponent which follows the number is also fetched by __fp_parse_round_after:wN.

```

12921 \cs_new:Npn \__fp_parse_small_round:NN #1#2
12922 {
12923     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
12924         +
12925         \exp_after:wN \__fp_round_s:NNNw
12926         \exp_after:wN 0
12927         \exp_after:wN #1
12928         \exp_after:wN #2
12929         \int_value:w \__fp_int_eval:w
12930         \exp_after:wN \__fp_parse_round_after:wN
12931         \int_value:w \__fp_int_eval:w 0 * \__fp_int_eval:w 0
12932         \exp_after:wN \__fp_parse_round_loop:N

```

```

12933         \exp:w
12934     \else:
12935         \__fp_parse_exponent:Nw #2
12936     \fi:
12937     \__fp_parse_expand:w
12938 }

```

(End definition for __fp_parse_small_round:NN and __fp_parse_round_after:wN.)

__fp_parse_large_round:NN Large numbers are harder to round, as there may be a period in the way. Again, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If there are no more digits (#2 is not a digit), then we must test for a period: if there is one, then switch to the rounding function for small significands, otherwise fetch an exponent. If there are more digits (#2 is a digit), then round, checking with __fp_parse_round_loop:N if all further digits vanish, or some are non-zero. This loop is not enough, as it is stopped by a period. After the loop, the aux function tests for a period: if it is present, then we must continue looking for digits, this time discarding the number of digits we find.

```

12939 \cs_new:Npn \__fp_parse_large_round:NN #1#2
12940 {
12941     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
12942     +
12943     \exp_after:wN \__fp_round_s:NNNw
12944     \exp_after:wN 0
12945     \exp_after:wN #1
12946     \exp_after:wN #2
12947     \int_value:w \__fp_int_eval:w
12948     \exp_after:wN \__fp_parse_large_round_aux:wNN
12949     \int_value:w \__fp_int_eval:w 1
12950     \exp_after:wN \__fp_parse_round_loop:N
12951 \else: %^^A could be dot, or e, or other
12952     \exp_after:wN \__fp_parse_large_round_test:NN
12953     \exp_after:wN #1
12954     \exp_after:wN #2
12955 \fi:
12956 }
12957 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
12958 {
12959     \if:w . \exp_not:N #2
12960         \exp_after:wN \__fp_parse_small_round:NN
12961         \exp_after:wN #1
12962         \exp:w
12963     \else:
12964         \__fp_parse_exponent:Nw #2
12965     \fi:
12966     \__fp_parse_expand:w
12967 }
12968 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
12969 {
12970     + #2
12971     \exp_after:wN \__fp_parse_round_after:wN
12972     \int_value:w \__fp_int_eval:w #1
12973     \if:w . \exp_not:N #3
12974         + 0 * \__fp_int_eval:w 0

```

```

12975         \exp_after:wN \__fp_parse_round_loop:N
12976         \exp:w \exp_after:wN \__fp_parse_expand:w
12977     \else:
12978         \exp_after:wN ;
12979         \exp_after:wN 0
12980         \exp_after:wN #3
12981     \fi:
12982 }

```

(End definition for `__fp_parse_large_round:NN`, `__fp_parse_large_round_test:NN`, and `__fp_parse_large_round_aux:wNN`.)

25.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.

```

\__fp_parse:n { 3.2 erf(0.1) }
\__fp_parse:n { 3.2 e\l_my_int }
\__fp_parse:n { 3.2 \c_pi_fp }

```

The first case indicates that just looking one character ahead for an “e” is not enough, since we would mistake the function `erf` for an exponent of “rf”. An alternative would be to look two tokens ahead and check if what follows is a sign or a digit, considering in that case that we must be finding an exponent. But taking care of the second case requires that we unpack registers after `e`. However, blindly expanding the two tokens ahead completely would break the third example (unpacking is even worse). Indeed, in the course of reading `3.2`, `\c_pi_fp` is expanded to `\s__fp __fp_chk:w 1 0 {-1} {3141} ...`; and `\s__fp` stops the expansion. Expanding two tokens ahead would then force the expansion of `__fp_chk:w` (despite it being protected), and that function tries to produce an error.

What can we do? Really, the reason why this last case breaks is that just as `TeX` does, we should read ahead as little as possible. Here, the only case where there may be an exponent is if the first token ahead is `e`. Then we expand (and possibly unpack) the second token.

`__fp_parse_exponent:Nw` This auxiliary is convenient to smuggle some material through `\fi:` ending conditional processing. We place those `\fi:` (argument #2) at a very odd place because this allows us to insert `__fp_int_eval:w ...` there if needed.

```

12983 \cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
12984 {
12985     \exp_after:wN ;
12986     \int_value:w #2 \__fp_parse_exponent:N #1
12987 }

```

(End definition for `__fp_parse_exponent:Nw`.)

`__fp_parse_exponent:N`
`__fp_parse_exponent_aux:N` This function should be called within an `\int_value:w` expansion (or within an integer expression). It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a semicolon. If there is no `e`, leave an exponent of 0. If there is an `e`, expand the next token to run some tests on it. The first rough test is that if the character code of #1 is greater than that of 9 (largest code valid for an exponent, less than any

code valid for an identifier), there was in fact no exponent; otherwise, we search for the sign of the exponent.

```

12988 \cs_new:Npn \__fp_parse_exponent:N #1
12989 {
12990   \if:w e \exp_not:N #1
12991     \exp_after:wN \__fp_parse_exponent_aux:N
12992     \exp:w
12993   \else:
12994     0 \__fp_parse_return_semicolon:w #1
12995   \fi:
12996   \__fp_parse_expand:w
12997 }
12998 \cs_new:Npn \__fp_parse_exponent_aux:N #1
12999 {
13000   \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #1
13001     0 \else: '#1 \fi: > '9 \exp_stop_f:
13002   0 \exp_after:wN ; \exp_after:wN e
13003   \else:
13004     \exp_after:wN \__fp_parse_exponent_sign:N
13005   \fi:
13006   #1
13007 }

```

(End definition for __fp_parse_exponent:N and __fp_parse_exponent_aux:N.)

__fp_parse_exponent_sign:N Read signs one by one (if there is any).

```

13008 \cs_new:Npn \__fp_parse_exponent_sign:N #1
13009 {
13010   \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
13011     \exp_after:wN \__fp_parse_exponent_sign:N
13012     \exp:w \exp_after:wN \__fp_parse_expand:w
13013   \else:
13014     \exp_after:wN \__fp_parse_exponent_body:N
13015     \exp_after:wN #1
13016   \fi:
13017 }

```

(End definition for __fp_parse_exponent_sign:N.)

__fp_parse_exponent_body:N An exponent can be an explicit integer (most common case), or various other things (most of which are invalid).

```

13018 \cs_new:Npn \__fp_parse_exponent_body:N #1
13019 {
13020   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
13021     \token_to_str:N #1
13022     \exp_after:wN \__fp_parse_exponent_digits:N
13023     \exp:w
13024   \else:
13025     \__fp_parse_exponent_keep:NTF #1
13026     { \__fp_parse_return_semicolon:w #1 }
13027     {
13028       \exp_after:wN ;
13029       \exp:w
13030     }

```

```

13031     \fi:
13032     \__fp_parse_expand:w
13033 }

```

(End definition for `__fp_parse_exponent_body:N`.)

`__fp_parse_exponent_digits:N` Read digits one by one, and leave them behind in the input stream. When finding a non-digit, stop, and insert a semicolon. Note that we do not check for overflow of the exponent, hence there can be a T_EX error. It is mostly harmless, except when parsing 0e9876543210, which should be a valid representation of 0, but is not.

```

13034 \cs_new:Npn \__fp_parse_exponent_digits:N #1
13035 {
13036   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
13037   \token_to_str:N #1
13038   \exp_after:wN \__fp_parse_exponent_digits:N
13039   \exp:w
13040   \else:
13041     \__fp_parse_return_semicolon:w #1
13042   \fi:
13043   \__fp_parse_expand:w
13044 }

```

(End definition for `__fp_parse_exponent_digits:N`.)

`__fp_parse_exponent_keep:N` This is the last building block for parsing exponents. The argument #1 is already fully expanded, and neither + nor - nor a digit. It can be:

- `\s__fp`, marking the start of an internal floating point, invalid here;
- another control sequence equal to `\relax`, probably a bad variable;
- a register: in this case we make sure that it is an integer register, not a dimension;
- a character other than +, - or digits, again, an error.

```

13045 \prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
13046 {
13047   \if_catcode:w \scan_stop: \exp_not:N #1
13048   \if_meaning:w \scan_stop: #1
13049   \if_int_compare:w
13050     \__fp_str_if_eq_x:nn { \s__fp } { \exp_not:N #1 }
13051     = 0 \exp_stop_f:
13052     0
13053     \__kernel_msg_expandable_error:nnn
13054     { kernel } { fp-after-e } { floating~point~ }
13055     \prg_return_true:
13056   \else:
13057     0
13058     \__kernel_msg_expandable_error:nnn
13059     { kernel } { bad-variable } { #1 }
13060     \prg_return_false:
13061   \fi:
13062   \else:
13063     \if_int_compare:w
13064       \__fp_str_if_eq_x:nn { \int_value:w #1 } { \tex_the:D #1 }
13065       = 0 \exp_stop_f:

```

```

13066         \int_value:w #1
13067     \else:
13068         0
13069         \__kernel_msg_expandable_error:nnn
13070         { kernel } { fp-after-e } { dimension~#1 }
13071     \fi:
13072     \prg_return_false:
13073 \fi:
13074 \else:
13075     0
13076     \__kernel_msg_expandable_error:nnn
13077     { kernel } { fp-missing } { exponent }
13078     \prg_return_true:
13079 \fi:
13080 }

```

(End definition for __fp_parse_exponent_keep:NTF.)

25.5 Constants, functions and prefix operators

25.5.1 Prefix operators

__fp_parse_prefix+:Nw A unary + does nothing: we should continue looking for a number.

```

13081 \cs_new_eq:cN { __fp_parse_prefix+:Nw } \__fp_parse_one:Nw

```

(End definition for __fp_parse_prefix+:Nw.)

_fp_parse_apply_function:NNNwN Here, #1 is a precedence, #2 is some extra data used by some functions, #3 is *e.g.*, __fp_sin_o:w, and expands once after the calculation, #4 is the operand, and #5 is a __fp_parse_infix...:N function. We feed the data #2, and the argument #4, to the function #3, which expands \exp:w thus the infix function #5.

```

13082 \cs_new:Npn \__fp_parse_apply_function:NNNwN #1#2#3#4#5
13083 {
13084     #3 #2 #4 @
13085     \exp:w \exp_end_continue_f:w #5 #1
13086 }

```

(End definition for __fp_parse_apply_function:NNNwN.)

_fp_parse_apply_unary:NNNwN In contrast to __fp_parse_apply_function:NNNwN, this checks that the operand #4 is a single argument (namely there is a single ;). We use the fact that any floating point starts with a “safe” token like \s__fp. If there is no argument produce the `fp-no-arg` error; if there are at least two produce `fp-multi-arg`. For the error message extract the mathematical function name (such as `sin`) from the `expl3` function that computes it, such as __fp_sin_o:w.

In addition, since there is a single argument we can dispatch on type and check that the resulting function exists. This catches things like `sin((1,2))` where it does not make sense to take the sine of a tuple.

```

13087 \cs_new:Npn \__fp_parse_apply_unary:NNNwN #1#2#3#4#5
13088 {
13089     \__fp_parse_apply_unary_chk:NwNw #4 @ ; . \q_stop
13090     \__fp_parse_apply_unary_type:NNN
13091     #3 #2 #4 @
13092     \exp:w \exp_end_continue_f:w #5 #1

```

```

13093 }
13094 \cs_new:Npn \__fp_parse_apply_unary_chk:NwNw #1#2 ; #3#4 \q_stop
13095 {
13096   \if_meaning:w @ #3 \else:
13097     \token_if_eq_meaning:NNTF . #3
13098     { \__fp_parse_apply_unary_chk:nNNNNw { no } }
13099     { \__fp_parse_apply_unary_chk:nNNNNw { multi } }
13100   \fi:
13101 }
13102 \cs_new:Npn \__fp_parse_apply_unary_chk:nNNNNw #1#2#3#4#5#6 @
13103 {
13104   #2
13105   \__fp_error:nffn { fp-#1-arg } { \__fp_func_to_name:N #4 } { } { }
13106   \exp_after:wN #4 \exp_after:wN #5 \c_nan_fp @
13107 }
13108 \cs_new:Npn \__fp_parse_apply_unary_type:NNN #1#2#3
13109 {
13110   \__fp_change_func_type:NNN #3 #1 \__fp_parse_apply_unary_error:NNw
13111   #2 #3
13112 }
13113 \cs_new:Npn \__fp_parse_apply_unary_error:NNw #1#2#3 @
13114 { \__fp_invalid_operation_o:fw { \__fp_func_to_name:N #1 } #3 }

```

(End definition for __fp_parse_apply_unary:NNNwN and others.)

__fp_parse_prefix_-:Nw
 __fp_parse_prefix_!:Nw

The unary - and boolean not are harder: we parse the operand using a precedence equal to the maximum of the previous precedence ##1 and the precedence \c__fp_prec_not_int of the unary operator, then call the appropriate __fp_⟨operation⟩_o:w function, where the ⟨operation⟩ is set_sign or not.

```

13115 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
13116 {
13117   \cs_new:cpn { \__fp_parse_prefix_ #1 :Nw } ##1
13118   {
13119     \exp_after:wN \__fp_parse_apply_unary:NNNwN
13120     \exp_after:wN ##1
13121     \exp_after:wN #4
13122     \exp_after:wN #3
13123     \exp:w
13124     \if_int_compare:w #2 < ##1
13125       \__fp_parse_operand:Nw ##1
13126     \else:
13127       \__fp_parse_operand:Nw #2
13128     \fi:
13129     \__fp_parse_expand:w
13130   }
13131 }
13132 \__fp_tmp:w - \c__fp_prec_not_int \__fp_set_sign_o:w 2
13133 \__fp_tmp:w ! \c__fp_prec_not_int \__fp_not_o:w ?

```

(End definition for __fp_parse_prefix_-:Nw and __fp_parse_prefix_!:Nw.)

__fp_parse_prefix_.:Nw

Numbers which start with a decimal separator (a period) end up here. Of course, we do not look for an operand, but for the rest of the number. This function is very similar to __fp_parse_one_digit:NN but calls __fp_parse_strim_zeros:N to trim zeros after

the decimal point, rather than the `trim_zeros` function for zeros before the decimal point.

```

13134 \cs_new:cpn { __fp_parse_prefix.:Nw } #1
13135 {
13136   \exp_after:wN \__fp_parse_infix_after_operand:NwN
13137   \exp_after:wN #1
13138   \exp:w \exp_end_continue_f:w
13139   \exp_after:wN \__fp_sanitizewN
13140   \int_value:w \__fp_int_eval:w 0 \__fp_parse_strim_zeros:N
13141 }

```

(End definition for `__fp_parse_prefix.:Nw`.)

`__fp_parse_prefix(:Nw`
`__fp_parse_lparen_after:NwN`

The left parenthesis is treated as a unary prefix operator because it appears in exactly the same settings. If the previous precedence is `\c__fp_prec_func_int` we are parsing arguments of a function and commas should not build tuples; otherwise commas should build tuples. We distinguish these cases by precedence: `\c__fp_prec_comma_int` for the case of arguments, `\c__fp_prec_tuple_int` for the case of tuples. Once the operand is found, the `lparen_after` auxiliary makes sure that there was a closing parenthesis (otherwise it complains), and leaves in the input stream an operand, fetching the following infix operator.

```

13142 \cs_new:cpn { __fp_parse_prefix(:Nw } #1
13143 {
13144   \exp_after:wN \__fp_parse_lparen_after:NwN
13145   \exp_after:wN #1
13146   \exp:w
13147   \if_int_compare:w #1 = \c__fp_prec_func_int
13148     \__fp_parse_operand:Nw \c__fp_prec_comma_int
13149   \else:
13150     \__fp_parse_operand:Nw \c__fp_prec_tuple_int
13151   \fi:
13152   \__fp_parse_expand:w
13153 }
13154 \cs_new:Npx \__fp_parse_lparen_after:NwN #1#2 @ #3
13155 {
13156   \exp_not:N \token_if_eq_meaning:NNTF #3
13157   \exp_not:c { __fp_parse_infix_):N }
13158   {
13159     \exp_not:N \__fp_exp_after_array_f:w #2 \s__fp_stop
13160     \exp_not:N \exp_after:wN
13161     \exp_not:N \__fp_parse_infix:NN
13162     \exp_not:N \exp_after:wN #1
13163     \exp_not:N \exp:w
13164     \exp_not:N \__fp_parse_expand:w
13165   }
13166   {
13167     \exp_not:N \__kernel_msg_expandable_error:nnn
13168     { kernel } { fp-missing } { ) }
13169     \exp_not:N \tl_if_empty:nT {#2} \exp_not:N \c__fp_empty_tuple_fp
13170     #2 @
13171     \exp_not:N \use_none:n #3
13172   }
13173 }

```

(End definition for _fp_parse_prefix_(:Nw and _fp_parse_lparen_after:NwN.)

_fp_parse_prefix_):Nw The right parenthesis can appear as a prefix in two similar cases: in an empty tuple or tuple ending with a comma, or in an empty argument list or argument list ending with a comma, such as in max(1,2,) or in rand().

```

13174 \cs_new:cpn { \_fp_parse_prefix_):Nw } #1
13175 {
13176   \if_int_compare:w #1 = \c__fp_prec_comma_int
13177   \else:
13178     \if_int_compare:w #1 = \c__fp_prec_tuple_int
13179     \exp_after:wN \c__fp_empty_tuple_fp \exp:w
13180   \else:
13181     \__kernel_msg_expandable_error:nnn
13182     { kernel } { fp-missing-number } { } }
13183   \exp_after:wN \c_nan_fp \exp:w
13184   \fi:
13185   \exp_end_continue_f:w
13186   \fi:
13187   \_fp_parse_infix:NN #1 )
13188 }
```

(End definition for _fp_parse_prefix_):Nw.)

25.5.2 Constants

_fp_parse_word_inf:N Some words correspond to constant floating points. The floating point constant is left as a result of _fp_parse_one:Nw after expanding _fp_parse_infix:NN.

```

13189 \cs_set_protected:Npn \_fp_tmp:w #1 #2
13190 {
13191   \cs_new:cpn { \_fp_parse_word_#1:N }
13192   { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \_fp_parse_infix:NN }
13193 }
13194 \_fp_tmp:w { inf } \c_inf_fp
13195 \_fp_tmp:w { nan } \c_nan_fp
13196 \_fp_tmp:w { pi } \c_pi_fp
13197 \_fp_tmp:w { deg } \c_one_degree_fp
13198 \_fp_tmp:w { true } \c_one_fp
13199 \_fp_tmp:w { false } \c_zero_fp
```

(End definition for _fp_parse_word_inf:N and others.)

_fp_parse_caseless_inf:N Copies of _fp_parse_word_...:N commands, to allow arbitrary case as mandated by the standard.

```

13200 \cs_new_eq:NN \_fp_parse_caseless_inf:N \_fp_parse_word_inf:N
13201 \cs_new_eq:NN \_fp_parse_caseless_infinity:N \_fp_parse_word_inf:N
13202 \cs_new_eq:NN \_fp_parse_caseless_nan:N \_fp_parse_word_nan:N
```

(End definition for _fp_parse_caseless_inf:N, _fp_parse_caseless_infinity:N, and _fp_parse_caseless_nan:N.)

_fp_parse_word_pt:N Dimension units are also floating point constants but their value is not stored as a floating point constant. We give the values explicitly here.

```

13203 \cs_set_protected:Npn \_fp_tmp:w #1 #2
13204 {
13205   \_fp_parse_word_pt:N
13206   \_fp_parse_word_in:N
13207   \_fp_parse_word_pc:N
13208   \_fp_parse_word_cm:N
13209   \_fp_parse_word_mm:N
13210   \_fp_parse_word_dd:N
13211   \_fp_parse_word_cc:N
13212   \_fp_parse_word_nd:N
13213   \_fp_parse_word_nc:N
13214   \_fp_parse_word_bp:N
13215   \_fp_parse_word_sp:N
```

```

13205 \cs_new:cpn { __fp_parse_word_#1:N }
13206 {
13207   \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
13208   \s__fp \__fp_chk:w 10 #2 ;
13209 }
13210 }
13211 \__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
13212 \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
13213 \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
13214 \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
13215 \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
13216 \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
13217 \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
13218 \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
13219 \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
13220 \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
13221 \__fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }

```

(End definition for __fp_parse_word_pt:N and others.)

__fp_parse_word_em:N The font-dependent units em and ex must be evaluated on the fly. We reuse an auxiliary of \dim_to_fp:n.

```

13222 \tl_map_inline:nn { {em} {ex} }
13223 {
13224   \cs_new:cpn { __fp_parse_word_#1:N }
13225   {
13226     \exp_after:wN \__fp_from_dim_test:ww
13227     \exp_after:wN 0 \exp_after:wN ,
13228     \int_value:w \dim_to_decimal_in_sp:n { 1 #1 } \exp_after:wN ;
13229     \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN
13230   }
13231 }

```

(End definition for __fp_parse_word_em:N and __fp_parse_word_ex:N.)

25.5.3 Functions

```

\__fp_parse_unary_function:NNN
\__fp_parse_function:NNN
13232 \cs_new:Npn \__fp_parse_unary_function:NNN #1#2#3
13233 {
13234   \exp_after:wN \__fp_parse_apply_unary:NNNwN
13235   \exp_after:wN #3
13236   \exp_after:wN #2
13237   \exp_after:wN #1
13238   \exp:w
13239   \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
13240 }
13241 \cs_new:Npn \__fp_parse_function:NNN #1#2#3
13242 {
13243   \exp_after:wN \__fp_parse_apply_function:NNNwN
13244   \exp_after:wN #3
13245   \exp_after:wN #2
13246   \exp_after:wN #1
13247   \exp:w

```

```

13248     \_fp_parse_operand:Nw \c\_fp_prec_func_int \_fp_parse_expand:w
13249 }

```

(End definition for _fp_parse_unary_function:NNN and _fp_parse_function:NNN.)

25.6 Main functions

_fp_parse:n Start an \exp:w expansion so that _fp_parse:n expands in two steps. The _fp_parse_operand:Nw function performs computations until reaching an operation with precedence \c_fp_prec_end_int or less, namely, the end of the expression. The marker \s_fp_mark indicates that the next token is an already parsed version of an infix operator, and _fp_parse_infix_end:N has infinitely negative precedence. Finally, clean up a (well-defined) set of extra tokens and stop the initial expansion with \exp_end:.

```

13250 \cs_new:Npn \_fp_parse:n #1
13251 {
13252   \exp:w
13253   \exp_after:wN \_fp_parse_after:ww
13254   \exp:w
13255   \_fp_parse_operand:Nw \c\_fp_prec_end_int
13256   \_fp_parse_expand:w #1
13257   \s\_fp_mark \_fp_parse_infix_end:N
13258   \s\_fp_stop
13259   \exp_end:
13260 }
13261 \cs_new:Npn \_fp_parse_after:ww
13262   #1@ \_fp_parse_infix_end:N \s\_fp_stop #2 { #2 #1 }
13263 \cs_new:Npn \_fp_parse_o:n #1
13264 {
13265   \exp:w
13266   \exp_after:wN \_fp_parse_after:ww
13267   \exp:w
13268   \_fp_parse_operand:Nw \c\_fp_prec_end_int
13269   \_fp_parse_expand:w #1
13270   \s\_fp_mark \_fp_parse_infix_end:N
13271   \s\_fp_stop
13272   {
13273     \exp_end_continue_f:w
13274     \_fp_exp_after_any_f:nw { \exp_after:wN \exp_stop_f: }
13275   }
13276 }

```

(End definition for _fp_parse:n, _fp_parse_o:n, and _fp_parse_after:ww.)

_fp_parse_operand:Nw This is just a shorthand which sets up both _fp_parse_continue:NwN and _fp_parse_one:Nw with the same precedence. Note the trailing \exp:w.

```

13277 \cs_new:Npn \_fp_parse_operand:Nw #1
13278 {
13279   \exp_end_continue_f:w
13280   \exp_after:wN \_fp_parse_continue:NwN
13281   \exp_after:wN #1
13282   \exp:w \exp_end_continue_f:w
13283   \exp_after:wN \_fp_parse_one:Nw
13284   \exp_after:wN #1
13285   \exp:w

```

```

13286     }
13287 \cs_new:Npn \__fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }

```

(End definition for __fp_parse_operand:Nw and __fp_parse_continue:NwN.)

__fp_parse_apply_binary:NwNwN Receives $\langle precedence \rangle \langle operand_1 \rangle @ \langle operation \rangle \langle operand_2 \rangle @ \langle infix command \rangle$. Builds the appropriate call to the $\langle operation \rangle$ #3, dispatching on both types. If the resulting control sequence does not exist, the operation is not allowed.

This is redefined in l3fp-extras.

```

13288 \cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2#3@ #4 #5#6@ #7
13289 {
13290     \exp_after:wN \__fp_parse_continue:NwN
13291     \exp_after:wN #1
13292     \exp:w \exp_end_continue_f:w
13293     \exp_after:wN \__fp_parse_apply_binary_chk:NN
13294     \cs:w
13295         __fp
13296         \__fp_type_from_scan:N #2
13297         _#4
13298         \__fp_type_from_scan:N #5
13299         _o:ww
13300     \cs_end:
13301     #4
13302     #2#3 #5#6
13303     \exp:w \exp_end_continue_f:w #7 #1
13304 }
13305 \cs_new:Npn \__fp_parse_apply_binary_chk:NN #1#2
13306 {
13307     \if_meaning:w \scan_stop: #1
13308     \__fp_parse_apply_binary_error:NNN #2
13309     \fi:
13310     #1
13311 }
13312 \cs_new:Npn \__fp_parse_apply_binary_error:NNN #1#2#3
13313 {
13314     #2
13315     \__fp_invalid_operation_o:Nww #1
13316 }

```

(End definition for __fp_parse_apply_binary:NwNwN, __fp_parse_apply_binary_chk:NN, and __fp_parse_apply_binary_error:NNN.)

__fp_binary_type_o:Nww Applies the operator #1 to its two arguments, dispatching according to their types, and expands once after the result. The rev version swaps its arguments before doing this.

```

13317 \cs_new:Npn \__fp_binary_type_o:Nww #1 #2#3 ; #4
13318 {
13319     \exp_after:wN \__fp_parse_apply_binary_chk:NN
13320     \cs:w
13321         __fp
13322         \__fp_type_from_scan:N #2
13323         _#1
13324         \__fp_type_from_scan:N #4
13325         _o:ww
13326     \cs_end:

```

```

13327         #1
13328         #2 #3 ; #4
13329     }
13330 \cs_new:Npn \__fp_binary_rev_type_o:Nww #1 #2#3 ; #4#5 ;
13331 {
13332     \exp_after:wN \__fp_parse_apply_binary_chk:NN
13333     \cs:w
13334         __fp
13335         \__fp_type_from_scan:N #4
13336         _ #1
13337         \__fp_type_from_scan:N #2
13338         _o:ww
13339     \cs_end:
13340     #1
13341     #4 #5 ; #2 #3 ;
13342 }

```

(End definition for __fp_binary_type_o:Nww and __fp_binary_rev_type_o:Nww.)

25.7 Infix operators

__fp_parse_infix_after_operand:NwN

```

13343 \cs_new:Npn \__fp_parse_infix_after_operand:NwN #1 #2;
13344 {
13345     \__fp_exp_after_f:nw { \__fp_parse_infix:NN #1 }
13346     #2;
13347 }
13348 \cs_new:Npn \__fp_parse_infix:NN #1 #2
13349 {
13350     \if_catcode:w \scan_stop: \exp_not:N #2
13351     \if_int_compare:w
13352         \__fp_str_if_eq_x:nn { \s__fp_mark } { \exp_not:N #2 }
13353         = 0 \exp_stop_f:
13354     \exp_after:wN \exp_after:wN
13355     \exp_after:wN \__fp_parse_infix_mark:NNN
13356     \else:
13357         \exp_after:wN \exp_after:wN
13358         \exp_after:wN \__fp_parse_infix_mul:N
13359     \fi:
13360     \else:
13361         \if_int_compare:w
13362             \__fp_int_eval:w
13363             ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
13364             = 3 \exp_stop_f:
13365         \exp_after:wN \exp_after:wN
13366         \exp_after:wN \__fp_parse_infix_mul:N
13367     \else:
13368         \exp_after:wN \__fp_parse_infix_check:NNN
13369         \cs:w
13370             __fp_parse_infix_ \token_to_str:N #2 :N
13371         \exp_after:wN \exp_after:wN \exp_after:wN
13372     \cs_end:
13373     \fi:
13374     \fi:

```

```

13375     #1
13376     #2
13377 }
13378 \cs_new:Npx \__fp_parse_infix_check:NNN #1#2#3
13379 {
13380     \exp_not:N \if_meaning:w \scan_stop: #1
13381     \exp_not:N \__kernel_msg_expandable_error:nnn
13382     { kernel } { fp-missing } { * }
13383     \exp_not:N \exp_after:wN
13384     \exp_not:c { \__fp_parse_infix_*:N }
13385     \exp_not:N \exp_after:wN #2
13386     \exp_not:N \exp_after:wN #3
13387     \exp_not:N \else:
13388     \exp_not:N \exp_after:wN #1
13389     \exp_not:N \exp_after:wN #2
13390     \exp_not:N \exp:w
13391     \exp_not:N \exp_after:wN
13392     \exp_not:N \__fp_parse_expand:w
13393     \exp_not:N \fi:
13394 }

```

(End definition for __fp_parse_infix_after_operand:NwN.)

25.7.1 Closing parentheses and commas

__fp_parse_infix_mark:NNN As an infix operator, __fp_mark means that the next token (#3) has already gone through __fp_parse_infix:NN and should be provided the precedence #1. The scan mark #2 is discarded.

```

13395 \cs_new:Npn \__fp_parse_infix_mark:NNN #1#2#3 { #3 #1 }

```

(End definition for __fp_parse_infix_mark:NNN.)

__fp_parse_infix_end:N This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```

13396 \cs_new:Npn \__fp_parse_infix_end:N #1
13397 { @ \use_none:n \__fp_parse_infix_end:N }

```

(End definition for __fp_parse_infix_end:N.)

__fp_parse_infix_):N This is very similar to __fp_parse_infix_end:N, complaining about an extra closing parenthesis if the previous operator was the beginning of the expression, with precedence \c__fp_prec_end_int.

```

13398 \cs_set_protected:Npn \__fp_tmp:w #1
13399 {
13400     \cs_new:Npn #1 ##1
13401     {
13402         \if_int_compare:w ##1 > \c__fp_prec_end_int
13403         \exp_after:wN @
13404         \exp_after:wN \use_none:n
13405         \exp_after:wN #1
13406     \else:
13407         \__kernel_msg_expandable_error:nnn { kernel } { fp-extra } { ) }
13408         \exp_after:wN \__fp_parse_infix:NN
13409         \exp_after:wN ##1

```

```

13410         \exp:w \exp_after:wN \__fp_parse_expand:w
13411     \fi:
13412 }
13413 }
13414 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_):N }

```

(End definition for __fp_parse_infix_):N.)

```

\__fp_parse_infix_,:N
\__fp_parse_infix_comma:w
\__fp_parse_apply_comma:NwNwN

```

As for other infix operations, if the previous operations has higher precedence the comma waits. Otherwise we call __fp_parse_operand:Nw to read more comma-delimited arguments that __fp_parse_infix_comma:w simply concatenates into a @-delimited array. The first comma in a tuple that is not a function argument is distinguished: in that case call __fp_parse_apply_comma:NwNwN whose job is to convert the first item of the tuple and an array of the remaining items into a tuple. In contrast to __fp_parse_apply_binary:NwNwN this function's operands are not single-object arrays.

```

13415 \cs_set_protected:Npn \__fp_tmp:w #1
13416 {
13417     \cs_new:Npn #1 ##1
13418     {
13419         \if_int_compare:w ##1 > \c__fp_prec_comma_int
13420             \exp_after:wN @
13421             \exp_after:wN \use_none:n
13422             \exp_after:wN #1
13423         \else:
13424             \if_int_compare:w ##1 < \c__fp_prec_comma_int
13425                 \exp_after:wN @
13426                 \exp_after:wN \__fp_parse_apply_comma:NwNwN
13427                 \exp_after:wN ,
13428                 \exp:w
13429             \else:
13430                 \exp_after:wN \__fp_parse_infix_comma:w
13431                 \exp:w
13432             \fi:
13433             \__fp_parse_operand:Nw \c__fp_prec_comma_int
13434             \exp_after:wN \__fp_parse_expand:w
13435         \fi:
13436     }
13437 }
13438 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_,:N }
13439 \cs_new:Npn \__fp_parse_infix_comma:w #1 @
13440 { #1 @ \use_none:n }
13441 \cs_new:Npn \__fp_parse_apply_comma:NwNwN #1 #2@ #3 #4@ #5
13442 {
13443     \exp_after:wN \__fp_parse_continue:NwN
13444     \exp_after:wN #1
13445     \exp:w \exp_end_continue_f:w
13446     \__fp_exp_after_tuple_f:nw { }
13447     \s__fp_tuple \__fp_tuple_chk:w { #2 #4 } ;
13448     #5 #1
13449 }

```

(End definition for __fp_parse_infix_,:N, __fp_parse_infix_comma:w, and __fp_parse_apply_comma:NwNwN.)

25.7.2 Usual infix operators

`__fp_parse_infix_+:N` As described in the “work plan”, each infix operator has an associated `\..._infix_...`
`__fp_parse_infix_-:N` function, a computing function, and precedence, given as arguments to `__fp_tmp:w`.
`__fp_parse_infix_/:N` Using the general mechanism for arithmetic operations. The power operation must be
`__fp_parse_infix_mul:N` associative in the opposite order from all others. For this, we use two distinct precedences.
`__fp_parse_infix_and:N`
`__fp_parse_infix_or:N`
`__fp_parse_infix_^:N`

```

13450 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
13451 {
13452   \cs_new:Npn #1 ##1
13453   {
13454     \if_int_compare:w ##1 < #3
13455       \exp_after:wN @
13456       \exp_after:wN \__fp_parse_apply_binary:NwNwN
13457       \exp_after:wN #2
13458       \exp:w
13459       \__fp_parse_operand:Nw #4
13460       \exp_after:wN \__fp_parse_expand:w
13461     \else:
13462       \exp_after:wN @
13463       \exp_after:wN \use_none:n
13464       \exp_after:wN #1
13465     \fi:
13466   }
13467 }
13468 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_^:N } ^
13469   \c__fp_prec_hatii_int \c__fp_prec_hat_int
13470 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_/:N } /
13471   \c__fp_prec_times_int \c__fp_prec_times_int
13472 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_mul:N } *
13473   \c__fp_prec_times_int \c__fp_prec_times_int
13474 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_-:N } -
13475   \c__fp_prec_plus_int \c__fp_prec_plus_int
13476 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_+:N } +
13477   \c__fp_prec_plus_int \c__fp_prec_plus_int
13478 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_and:N } &
13479   \c__fp_prec_and_int \c__fp_prec_and_int
13480 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_or:N } |
13481   \c__fp_prec_or_int \c__fp_prec_or_int

```

(End definition for `__fp_parse_infix_+:N` and others.)

25.7.3 Juxtaposition

`__fp_parse_infix_(:N` When an opening parenthesis appears where we expect an infix operator, we compute
the product of the previous operand and the contents of the parentheses using `__fp_-
parse_infix_mul:N`.

```

13482 \cs_new:cpn { \__fp_parse_infix_(:N } #1
13483 { \__fp_parse_infix_mul:N #1 ( }

```

(End definition for `__fp_parse_infix_(:N`.)

25.7.4 Multi-character cases

_fp_parse_infix_*:N

```

13484 \cs_set_protected:Npn \_fp_tmp:w #1
13485 {
13486   \cs_new:cpn { \_fp_parse_infix_*:N } ##1##2
13487   {
13488     \if:w * \exp_not:N ##2
13489       \exp_after:wN #1
13490       \exp_after:wN ##1
13491     \else:
13492       \exp_after:wN \_fp_parse_infix_mul:N
13493       \exp_after:wN ##1
13494       \exp_after:wN ##2
13495     \fi:
13496   }
13497 }
13498 \exp_args:Nc \_fp_tmp:w { \_fp_parse_infix_~:N }

```

(End definition for _fp_parse_infix_*:N.)

_fp_parse_infix_|:Nw

_fp_parse_infix_&:Nw

```

13499 \cs_set_protected:Npn \_fp_tmp:w #1#2#3
13500 {
13501   \cs_new:Npn #1 ##1##2
13502   {
13503     \if:w #2 \exp_not:N ##2
13504       \exp_after:wN #1
13505       \exp_after:wN ##1
13506       \exp:w \exp_after:wN \_fp_parse_expand:w
13507     \else:
13508       \exp_after:wN #3
13509       \exp_after:wN ##1
13510       \exp_after:wN ##2
13511     \fi:
13512   }
13513 }
13514 \exp_args:Nc \_fp_tmp:w { \_fp_parse_infix_|:N } | \_fp_parse_infix_or:N
13515 \exp_args:Nc \_fp_tmp:w { \_fp_parse_infix_&:N } & \_fp_parse_infix_and:N

```

(End definition for _fp_parse_infix_|:Nw and _fp_parse_infix_&:Nw.)

25.7.5 Ternary operator

_fp_parse_infix_?:N

_fp_parse_infix_:N

```

13516 \cs_set_protected:Npn \_fp_tmp:w #1#2#3#4
13517 {
13518   \cs_new:Npn #1 ##1
13519   {
13520     \if_int_compare:w ##1 < \c__fp_prec_quest_int
13521       #4
13522       \exp_after:wN @
13523       \exp_after:wN #2
13524     \exp:w

```

```

13525         \__fp_parse_operand:Nw #3
13526         \exp_after:wN \__fp_parse_expand:w
13527     \else:
13528         \exp_after:wN @
13529         \exp_after:wN \use_none:n
13530         \exp_after:wN #1
13531     \fi:
13532 }
13533 }
13534 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_?:N }
13535 \__fp_ternary:NwwN \c__fp_prec_quest_int { }
13536 \exp_args:Nc \__fp_tmp:w { \__fp_parse_infix_::N }
13537 \__fp_ternary_auxii:NwwN \c__fp_prec_colon_int
13538 {
13539     \__kernel_msg_expandable_error:nnnn
13540     { kernel } { fp-missing } { ? } { ~for~?: }
13541 }

```

(End definition for __fp_parse_infix_?:N and __fp_parse_infix_::N.)

25.7.6 Comparisons

```

\__fp_parse_infix_<:N
\__fp_parse_infix_=:N
\__fp_parse_infix_>:N
\__fp_parse_infix_!:N
\__fp_parse_excl_error:
\__fp_parse_compare:NNNNNNN
\__fp_parse_compare_auxi:NNNNNNN
\__fp_parse_compare_auxii:NNNNN
\__fp_parse_compare_end:NNNNw
\__fp_compare:wNNNNw
13542 \cs_new:cpn { \__fp_parse_infix_<:N } #1
13543 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 < }
13544 \cs_new:cpn { \__fp_parse_infix_=:N } #1
13545 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 = }
13546 \cs_new:cpn { \__fp_parse_infix_>:N } #1
13547 { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 > }
13548 \cs_new:cpn { \__fp_parse_infix_!:N } #1
13549 {
13550     \exp_after:wN \__fp_parse_compare:NNNNNNN
13551     \exp_after:wN #1
13552     \exp_after:wN 0
13553     \exp_after:wN 1
13554     \exp_after:wN 1
13555     \exp_after:wN 1
13556     \exp_after:wN 1
13557 }
13558 \cs_new:Npn \__fp_parse_excl_error:
13559 {
13560     \__kernel_msg_expandable_error:nnnn
13561     { kernel } { fp-missing } { = } { ~after~!. }
13562 }
13563 \cs_new:Npn \__fp_parse_compare:NNNNNNN #1
13564 {
13565     \if_int_compare:w #1 < \c__fp_prec_comp_int
13566         \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
13567         \exp_after:wN \__fp_parse_excl_error:
13568     \else:
13569         \exp_after:wN @
13570         \exp_after:wN \use_none:n
13571         \exp_after:wN \__fp_parse_compare:NNNNNNN
13572     \fi:

```

```

13573     }
13574 \cs_new:Npn \__fp_parse_compare_auxi:NNNNNN #1#2#3#4#5#6#7
13575 {
13576     \if_case:w
13577         \__fp_int_eval:w \exp_after:wN ' \token_to_str:N #7 - '<
13578         \__fp_int_eval_end:
13579         \__fp_parse_compare_auxii:NNNN #2#2#4#5#6
13580     \or: \__fp_parse_compare_auxii:NNNN #2#3#2#5#6
13581     \or: \__fp_parse_compare_auxii:NNNN #2#3#4#2#6
13582     \or: \__fp_parse_compare_auxii:NNNN #2#3#4#5#2
13583     \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
13584     \fi:
13585 }
13586 \cs_new:Npn \__fp_parse_compare_auxii:NNNN #1#2#3#4#5
13587 {
13588     \exp_after:wN \__fp_parse_compare_auxi:NNNNNN
13589     \exp_after:wN \prg_do_nothing:
13590     \exp_after:wN #1
13591     \exp_after:wN #2
13592     \exp_after:wN #3
13593     \exp_after:wN #4
13594     \exp_after:wN #5
13595     \exp:w \exp_after:wN \__fp_parse_expand:w
13596 }
13597 \cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
13598 {
13599     \fi:
13600     \exp_after:wN @
13601     \exp_after:wN \__fp_parse_apply_compare:NwNNNNNNwN
13602     \exp_after:wN \c_one_fp
13603     \exp_after:wN #1
13604     \exp_after:wN #2
13605     \exp_after:wN #3
13606     \exp_after:wN #4
13607     \exp:w
13608     \__fp_parse_operand:Nw \c__fp_prec_comp_int \__fp_parse_expand:w #5
13609 }
13610 \cs_new:Npn \__fp_parse_apply_compare:NwNNNNNNwN
13611     #1 #2@ #3 #4#5#6#7 #8@ #9
13612 {
13613     \if_int_odd:w
13614         \if_meaning:w \c_zero_fp #3
13615         0
13616     \else:
13617         \if_case:w \__fp_compare_back_any:ww #8 #2 \exp_stop_f:
13618             #5 \or: #6 \or: #7 \else: #4
13619         \fi:
13620     \fi:
13621     \exp_stop_f:
13622     \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
13623     \exp_after:wN \c_one_fp
13624 \else:
13625     \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
13626     \exp_after:wN \c_zero_fp

```

```

13627     \fi:
13628     #1 #8 #9
13629   }
13630 \cs_new:Npn \__fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
13631 {
13632   \if_meaning:w \__fp_parse_compare:NNNNNN #4
13633     \exp_after:wN \__fp_parse_continue_compare:NNwNN
13634     \exp_after:wN #1
13635     \exp_after:wN #2
13636     \exp:w \exp_end_continue_f:w
13637     \__fp_exp_after_o:w #3;
13638     \exp:w \exp_end_continue_f:w
13639   \else:
13640     \exp_after:wN \__fp_parse_continue:NwN
13641     \exp_after:wN #2
13642     \exp:w \exp_end_continue_f:w
13643     \exp_after:wN #1
13644     \exp:w \exp_end_continue_f:w
13645   \fi:
13646   #4 #2
13647 }
13648 \cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
13649 { #4 #2 #3@ #1 }

```

(End definition for __fp_parse_infix_<:N and others.)

25.8 Tools for functions

__fp_parse_function_all_fp_o:fnw Followed by $\{\langle function\ name\rangle\}\{\langle code\rangle\}\langle float\ array\rangle$ @ this checks all floats are floating point numbers (no tuples).

```

13650 \cs_new:Npn \__fp_parse_function_all_fp_o:fnw #1#2#3 @
13651 {
13652   \__fp_array_if_all_fp:nTF {#3}
13653   { #2 #3 @ }
13654   {
13655     \__fp_error:nffn { fp-bad-args }
13656     {#1}
13657     { \fp_to_tl:n { \s__fp_tuple \__fp_tuple_chk:w {#3} ; } }
13658     { }
13659     \exp_after:wN \c_nan_fp
13660   }
13661 }

```

(End definition for __fp_parse_function_all_fp_o:fnw.)

__fp_parse_function_one_two:nnw This is followed by $\{\langle function\ name\rangle\}\{\langle code\rangle\}\langle float\ array\rangle$ @. It checks that the $\langle float\ array\rangle$ consists of one or two floating point numbers (not tuples), then leaves the $\langle code\rangle$ (if there is one float) or its tail (if there are two floats) followed by the $\langle float\ array\rangle$. The $\langle code\rangle$ should start with a single token such as __fp_atan_default:w that deals with the single-float case.

The first __fp_if_type_fp:NTwFw test catches the case of no argument and the case of a tuple argument. The next one distinguishes the case of a single argument (no error, just add \c_one_fp) from a tuple second argument. Finally check there is no further argument.

```

13662 \cs_new:Npn \__fp_parse_function_one_two:nnw #1#2#3
13663 {
13664   \__fp_if_type_fp:NTwFw
13665   #3 { } \s__fp \__fp_parse_function_one_two_error_o:w \q_stop
13666   \__fp_parse_function_one_two_aux:nnw {#1} {#2} #3
13667 }
13668 \cs_new:Npn \__fp_parse_function_one_two_error_o:w #1#2#3#4 @
13669 {
13670   \__fp_error:nffn { fp-bad-args }
13671   {#2}
13672   { \fp_to_tl:n { \s__fp_tuple \__fp_tuple_chk:w {#4} ; } }
13673   { }
13674   \exp_after:wN \c_nan_fp
13675 }
13676 \cs_new:Npn \__fp_parse_function_one_two_aux:nnw #1#2 #3; #4
13677 {
13678   \__fp_if_type_fp:NTwFw
13679   #4 { }
13680   \s__fp
13681   {
13682     \if_meaning:w @ #4
13683     \exp_after:wN \use_iv:nnnn
13684     \fi:
13685     \__fp_parse_function_one_two_error_o:w
13686   }
13687   \q_stop
13688   \__fp_parse_function_one_two_auxii:nnw {#1} {#2} #3; #4
13689 }
13690 \cs_new:Npn \__fp_parse_function_one_two_auxii:nnw #1#2#3; #4; #5
13691 {
13692   \if_meaning:w @ #5 \else:
13693     \exp_after:wN \__fp_parse_function_one_two_error_o:w
13694     \fi:
13695     \use_ii:nn {#1} { \use_none:n #2 } #3; #4; #5
13696 }

```

(End definition for __fp_parse_function_one_two:nnw and others.)

__fp_tuple_map_o:nw Apply #1 to all items in the following tuple and expand once afterwards. The code #1
 __fp_tuple_map_loop_o:nw should itself expand once after its result.

```

13697 \cs_new:Npn \__fp_tuple_map_o:nw #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
13698 {
13699   \exp_after:wN \s__fp_tuple
13700   \exp_after:wN \__fp_tuple_chk:w
13701   \exp_after:wN {
13702     \exp:w \exp_end_continue_f:w
13703     \__fp_tuple_map_loop_o:nw {#1} #2
13704     { \s__fp \prg_break: } ;
13705     \prg_break_point:
13706     \exp_after:wN } \exp_after:wN ;
13707   }
13708 \cs_new:Npn \__fp_tuple_map_loop_o:nw #1#2#3 ;
13709 {
13710   \use_none:n #2

```

```

13711     #1 #2 #3 ;
13712     \exp:w \exp_end_continue_f:w
13713     \__fp_tuple_map_loop_o:nw {#1}
13714 }

```

(End definition for __fp_tuple_map_o:nw and __fp_tuple_map_loop_o:nw.)

__fp_tuple_mapthread_o:nww Apply #1 to pairs of items in the two following tuples and expand once afterwards.

```

\__fp_tuple_mapthread_loop_o:nw
13715 \cs_new:Npn \__fp_tuple_mapthread_o:nww #1
13716     \s__fp_tuple \__fp_tuple_chk:w #2 ;
13717     \s__fp_tuple \__fp_tuple_chk:w #3 ;
13718 {
13719     \exp_after:wN \s__fp_tuple
13720     \exp_after:wN \__fp_tuple_chk:w
13721     \exp_after:wN {
13722         \exp:w \exp_end_continue_f:w
13723         \__fp_tuple_mapthread_loop_o:nw {#1}
13724         #2 { \s__fp \prg_break: } ; @
13725         #3 { \s__fp \prg_break: } ;
13726         \prg_break_point:
13727     \exp_after:wN } \exp_after:wN ;
13728 }
13729 \cs_new:Npn \__fp_tuple_mapthread_loop_o:nw #1#2#3 ; #4 @ #5#6 ;
13730 {
13731     \use_none:n #2
13732     \use_none:n #5
13733     #1 #2 #3 ; #5 #6 ;
13734     \exp:w \exp_end_continue_f:w
13735     \__fp_tuple_mapthread_loop_o:nw {#1} #4 @
13736 }

```

(End definition for __fp_tuple_mapthread_o:nww and __fp_tuple_mapthread_loop_o:nw.)

25.9 Candidate: defining new l3fp functions

\fp_function:Nw Parse the argument of the function #1 using __fp_parse_operand:Nw with a precedence of 16, and pass the function and argument to __fp_function_apply:nw.

```

13737 \cs_new:Npn \fp_function:Nw #1
13738 {
13739     \exp_after:wN \__fp_function_apply:nw
13740     \exp_after:wN #1
13741     \exp:w
13742     \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
13743 }

```

(End definition for \fp_function:Nw. This function is documented on page ??.)

\fp_new_function:Npn Save the code provided by the user in the control sequence __fp_user_#1. Define __fp_new_function:NNnnn #1 to call __fp_function_apply:nw after parsing one operand using __fp_parse_operand:Nw with precedence 16. The auxiliary __fp_function_args:Nwn receives the user function and the number of arguments (half of the number of tokens in the parameter text #2), followed by the operand (as a token list of floating points). It checks the number of arguments, and applies the user function to the arguments (without the outer brace group).

```

13744 \cs_new_protected:Npn \fp_new_function:Npn #1#2#
13745 {
13746   \__fp_new_function:Ncfnn #1
13747   { \__fp_user_ \cs_to_str:N #1 }
13748   { \int_eval:n { \tl_count:n {#2} / 2 } }
13749   {#2}
13750 }
13751 \cs_new_protected:Npn \__fp_new_function:NNnnn #1#2#3#4#5
13752 {
13753   \cs_new:Npn #1
13754   {
13755     \exp_after:wN \__fp_function_apply:nw \exp_after:wN
13756     {
13757       \exp_after:wN \__fp_function_args:Nwn
13758       \exp_after:wN #2
13759       \int_value:w #3 \exp_after:wN ; \exp_after:wN
13760     }
13761     \exp:w
13762     \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
13763   }
13764   \cs_new:Npn #2 #4 {#5}
13765 }
13766 \cs_generate_variant:Nn \__fp_new_function:NNnnn { Ncf }
13767 \cs_new:Npn \__fp_function_args:Nwn #1#2; #3
13768 {
13769   \int_compare:nNnTF { \tl_count:n {#3} } = {#2}
13770   { #1 #3 }
13771   {
13772     \__kernel_msg_expandable_error:nnnnn
13773     { kernel } { fp-num-args } { #1() } {#2} {#2}
13774     \c_nan_fp
13775   }
13776 }

```

(End definition for \fp_new_function:Npn, __fp_new_function:NNnnn, and __fp_function_args:Nwn. This function is documented on page ??.)

```

\__fp_function_apply:nw
\__fp_function_store:wwNwnn
  \__fp_function_store_end:wnnn

```

The auxiliary __fp_function_apply:nw is called after parsing an operand, so it receives some code #1, then the operand ending with @, then a function such as __fp_parse_infix_+:N (but not always of this form, see comparisons for instance). Package the operand (an array) into a token list with floating point items: this is the role of __fp_function_store:wwNwnn and __fp_function_store_end:wnnn. Then apply __fp_parse:n to the code #1 followed by a brace group with this token list. This results in a floating point result, which is then correctly parsed as the next operand of whatever was looking for one. The trailing \s__fp_mark is used as a special infix operator to indicate that the next token has already gone through __fp_parse_infix:NN.

```

13777 \cs_new:Npn \__fp_function_apply:nw #1#2 @
13778 {
13779   \__fp_parse:n
13780   {
13781     \__fp_function_store:wwNwnn #2
13782     \s__fp_mark \__fp_function_store:wwNwnn ;
13783     \s__fp_mark \__fp_function_store_end:wnnn
13784     \s__fp_stop { } { } {#1}

```



```

13785     }
13786     \s__fp_mark
13787   }
13788   \cs_new:Npn \__fp_function_store:wwNwnn
13789     #1; #2 \s__fp_mark #3#4 \s__fp_stop #5#6
13790     { #3 #2 \s__fp_mark #3#4 \s__fp_stop { #5 #6 } { { #1; } } }
13791   \cs_new:Npn \__fp_function_store_end:wnnn
13792     #1 \s__fp_stop #2#3#4
13793     { #4 {#2} }

```

(End definition for __fp_function_apply:nw, __fp_function_store:wwNwnn, and __fp_function_store_end:wnnn.)

25.10 Messages

```

13794 \__kernel_msg_new:nnn { kernel } { fp-deprecated }
13795   { '#1'~deprecated;~use~'#2' }
13796 \__kernel_msg_new:nnn { kernel } { unknown-fp-word }
13797   { Unknown~fp~word~#1. }
13798 \__kernel_msg_new:nnn { kernel } { fp-missing }
13799   { Missing~#1~inserted #2. }
13800 \__kernel_msg_new:nnn { kernel } { fp-extra }
13801   { Extra~#1~ignored. }
13802 \__kernel_msg_new:nnn { kernel } { fp-early-end }
13803   { Premature~end~in~fp~expression. }
13804 \__kernel_msg_new:nnn { kernel } { fp-after-e }
13805   { Cannot~use~#1 after~'e'. }
13806 \__kernel_msg_new:nnn { kernel } { fp-missing-number }
13807   { Missing~number~before~'#1'. }
13808 \__kernel_msg_new:nnn { kernel } { fp-unknown-symbol }
13809   { Unknown~symbol~#1~ignored. }
13810 \__kernel_msg_new:nnn { kernel } { fp-extra-comma }
13811   { Unexpected~comma~turned~to~nan~result. }
13812 \__kernel_msg_new:nnn { kernel } { fp-no-arg }
13813   { #1~got~no~argument;~used~nan. }
13814 \__kernel_msg_new:nnn { kernel } { fp-multi-arg }
13815   { #1~got~more~than~one~argument;~used~nan. }
13816 \__kernel_msg_new:nnn { kernel } { fp-num-args }
13817   { #1~expects~between~#2~and~#3~arguments. }
13818 \__kernel_msg_new:nnn { kernel } { fp-bad-args }
13819   { Arguments~in~#1#2~are~invalid. }
13820 \__kernel_msg_new:nnn { kernel } { fp-infty-pi }
13821   { Math~command~#1 is~not~an~fp }
13822 <*package>
13823 \cs_if_exist:cT { @unexpandable@protect }
13824   {
13825     \__kernel_msg_new:nnn { kernel } { fp-robust-cmd }
13826     { Robust~command~#1 invalid~in~fp~expression! }
13827   }
13828 </package>
13829 </initex | package>

```

26 l3fp-assign implementation

13830 <*initex | package>

13831 <@@=fp>

26.1 Assigning values

\fp_new:N Floating point variables are initialized to be +0.

```
13832 \cs_new_protected:Npn \fp_new:N #1
13833 { \cs_new_eq:NN #1 \c_zero_fp }
13834 \cs_generate_variant:Nn \fp_new:N {c}
```

(End definition for \fp_new:N. This function is documented on page 179.)

\fp_set:Nn Simply use __fp_parse:n within various f-expanding assignments.

```
\fp_set:cn 13835 \cs_new_protected:Npn \fp_set:Nn #1#2
\fp_gset:Nn 13836 { \tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_gset:cn 13837 \cs_new_protected:Npn \fp_gset:Nn #1#2
\fp_const:Nn 13838 { \tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_const:cn 13839 \cs_new_protected:Npn \fp_const:Nn #1#2
13840 { \tl_const:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
13841 \cs_generate_variant:Nn \fp_set:Nn {c}
13842 \cs_generate_variant:Nn \fp_gset:Nn {c}
13843 \cs_generate_variant:Nn \fp_const:Nn {c}
```

(End definition for \fp_set:Nn, \fp_gset:Nn, and \fp_const:Nn. These functions are documented on page 180.)

\fp_set_eq:NN Copying a floating point is the same as copying the underlying token list.

```
\fp_set_eq:cn 13844 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
\fp_set_eq:Nc 13845 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\fp_set_eq:cc 13846 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
\fp_gset_eq:NN 13847 \cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }
```

(End definition for \fp_set_eq:NN and \fp_gset_eq:NN. These functions are documented on page 180.)

\fp_set_eq:cn Setting a floating point to zero: copy \c_zero_fp.

```
\fp_zero:c 13848 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 13849 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 13850 \cs_generate_variant:Nn \fp_zero:N { c }
13851 \cs_generate_variant:Nn \fp_gzero:N { c }
```

(End definition for \fp_zero:N and \fp_gzero:N. These functions are documented on page 179.)

\fp_zero_new:N Set the floating point to zero, or define it if needed.

```
\fp_zero_new:c 13852 \cs_new_protected:Npn \fp_zero_new:N #1
\fp_gzero_new:N 13853 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:c 13854 \cs_new_protected:Npn \fp_gzero_new:N #1
13855 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
13856 \cs_generate_variant:Nn \fp_zero_new:N { c }
13857 \cs_generate_variant:Nn \fp_gzero_new:N { c }
```

(End definition for \fp_zero_new:N and \fp_gzero_new:N. These functions are documented on page 179.)

26.2 Updating values

These match the equivalent functions in `l3int` and `l3skip`.

`\fp_add:Nn` For the sake of error recovery we should not simply set `#1` to `#1 ± (#2)`: for instance, if `#2` is `0)+2`, the parsing error would be raised at the last closing parenthesis rather than at the closing parenthesis in the user argument. Thus we evaluate `#2` instead of just putting parentheses. As an optimization we use `__fp_parse:n` rather than `\fp_eval:n`, which would convert the result away from the internal representation and back.

`\fp_add:cn`

`\fp_gadd:Nn`

`\fp_gadd:cn`

`\fp_sub:Nn`

`\fp_sub:cn`

`\fp_gsub:Nn`

`\fp_gsub:cn`

`__fp_add:NNNn`

```

13858 \cs_new_protected:Npn \fp_add:Nn { \__fp_add:NNNn \fp_set:Nn + }
13859 \cs_new_protected:Npn \fp_gadd:Nn { \__fp_add:NNNn \fp_gset:Nn + }
13860 \cs_new_protected:Npn \fp_sub:Nn { \__fp_add:NNNn \fp_set:Nn - }
13861 \cs_new_protected:Npn \fp_gsub:Nn { \__fp_add:NNNn \fp_gset:Nn - }
13862 \cs_new_protected:Npn \__fp_add:NNNn #1#2#3#4
13863 { #1 #3 { #3 #2 \__fp_parse:n {#4} } }
13864 \cs_generate_variant:Nn \fp_add:Nn { c }
13865 \cs_generate_variant:Nn \fp_gadd:Nn { c }
13866 \cs_generate_variant:Nn \fp_sub:Nn { c }
13867 \cs_generate_variant:Nn \fp_gsub:Nn { c }

```

(End definition for `\fp_add:Nn` and others. These functions are documented on page 180.)

26.3 Showing values

`\fp_show:N` This shows the result of computing its argument by passing the right data to `\tl_show:n` or `\tl_log:n`.

`\fp_show:c`

`\fp_log:N`

`\fp_log:c`

`__fp_show:NN`

```

13868 \cs_new_protected:Npn \fp_show:N { \__fp_show:NN \tl_show:n }
13869 \cs_generate_variant:Nn \fp_show:N { c }
13870 \cs_new_protected:Npn \fp_log:N { \__fp_show:NN \tl_log:n }
13871 \cs_generate_variant:Nn \fp_log:N { c }
13872 \cs_new_protected:Npn \__fp_show:NN #1#2
13873 {
13874   \__kernel_chk_defined:NT #2
13875   { \exp_args:Nx #1 { \token_to_str:N #2 = \fp_to_tl:N #2 } }
13876 }

```

(End definition for `\fp_show:N`, `\fp_log:N`, and `__fp_show:NN`. These functions are documented on page 187.)

`\fp_show:n` Use general tools.

`\fp_log:n`

```

13877 \cs_new_protected:Npn \fp_show:n
13878 { \msg_show_eval:Nn \fp_to_tl:n }
13879 \cs_new_protected:Npn \fp_log:n
13880 { \msg_log_eval:Nn \fp_to_tl:n }

```

(End definition for `\fp_show:n` and `\fp_log:n`. These functions are documented on page 187.)

26.4 Some useful constants and scratch variables

`\c_one_fp` Some constants.

`\c_e_fp`

```

13881 \fp_const:Nn \c_e_fp { 2.718 2818 2845 9045 }
13882 \fp_const:Nn \c_one_fp { 1 }

```

(End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 185.)

`\c_pi_fp` We simply round π to and $\pi/180$ to 16 significant digits.
`\c_one_degree_fp` 13883 `\fp_const:Nn \c_pi_fp { 3.141 5926 5358 9793 }`
13884 `\fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }`
(End definition for `\c_pi_fp` and `\c_one_degree_fp`. These variables are documented on page 186.)

`\l_tmpa_fp` Scratch variables are simply initialized there.
`\l_tmpb_fp` 13885 `\fp_new:N \l_tmpa_fp`
`\g_tmpa_fp` 13886 `\fp_new:N \l_tmpb_fp`
`\g_tmpb_fp` 13887 `\fp_new:N \g_tmpa_fp`
13888 `\fp_new:N \g_tmpb_fp`
(End definition for `\l_tmpa_fp` and others. These variables are documented on page 186.)
13889 `\</initex | package>`

27 l3fp-logic Implementation

13890 `*initex | package>`
13891 `\<@@=fp>`
`__fp_parse_word_max:N` Those functions may receive a variable number of arguments.
`__fp_parse_word_min:N` 13892 `\cs_new:Npn __fp_parse_word_max:N`
13893 `{ __fp_parse_function:NNN __fp_minmax_o:Nw 2 }`
13894 `\cs_new:Npn __fp_parse_word_min:N`
13895 `{ __fp_parse_function:NNN __fp_minmax_o:Nw 0 }`
(End definition for `__fp_parse_word_max:N` and `__fp_parse_word_min:N`.)

27.1 Syntax of internal functions

- `__fp_compare_npos:nwnw {<expo1>} <body1> ; {<expo2>} <body2> ;`
- `__fp_minmax_o:Nw <sign> <floating point array>`
- `__fp_not_o:w ? <floating point array>` (with one floating point number only)
- `__fp_&_o:ww <floating point> <floating point>`
- `__fp_|_o:ww <floating point> <floating point>`
- `__fp_ternary:NwwN, __fp_ternary_auxi:NwwN, __fp_ternary_auxii:NwwN` have to be understood.

27.2 Existence test

`\fp_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.
`\fp_if_exist_p:c` 13896 `\prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }`
`\fp_if_exist:NTF` 13897 `\prg_new_eq_conditional:NNn \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }`
`\fp_if_exist:cTF` (End definition for `\fp_if_exist:NTF`. This function is documented on page 182.)

27.3 Comparison

`\fp_compare_p:n` Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with ± 0 . Tuples are true.

`\fp_compare:nTF`

```

13898 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
13899 {
13900   \exp_after:wN \__fp_compare_return:w
13901   \exp:w \exp_end_continue_f:w \__fp_parse:n {#1}
13902 }
13903 \cs_new:Npn \__fp_compare_return:w #1#2#3;
13904 {
13905   \if_charcode:w 0
13906     \__fp_if_type_fp:NTwFw
13907     #1 { \use_i_delimit_by_q_stop:nw #3 \q_stop }
13908     \s_fp 1 \q_stop
13909     \prg_return_false:
13910   \else:
13911     \prg_return_true:
13912   \fi:
13913 }
```

(End definition for `\fp_compare:nTF` and `__fp_compare_return:w`. This function is documented on page 183.)

`\fp_compare_p:nNn`

`\fp_compare:nNnTF`

`__fp_compare_aux:wn`

Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point numbers swapped to `__fp_compare_back_any:ww`, defined below. Compare the result with '#2-=' , which is -1 for $<$, 0 for $=$, 1 for $>$ and 2 for $?$.

```

13914 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
13915 {
13916   \if_int_compare:w
13917     \exp_after:wN \__fp_compare_aux:wn
13918     \exp:w \exp_end_continue_f:w \__fp_parse:n {#1} {#3}
13919     = \__fp_int_eval:w '#2 - '=' \__fp_int_eval_end:
13920     \prg_return_true:
13921   \else:
13922     \prg_return_false:
13923   \fi:
13924 }
13925 \cs_new:Npn \__fp_compare_aux:wn #1; #2
13926 {
13927   \exp_after:wN \__fp_compare_back_any:ww
13928   \exp:w \exp_end_continue_f:w \__fp_parse:n {#2} #1;
13929 }
```

(End definition for `\fp_compare:nNnTF` and `__fp_compare_aux:wn`. This function is documented on page 182.)

`__fp_compare_back_any:ww`

`__fp_compare_back:ww`

`__fp_compare_nan:w`

`__fp_compare_back_any:ww` $\langle y \rangle$; $\langle x \rangle$;

Expands (in the same way as `\int_eval:n`) to -1 if $x < y$, 0 if $x = y$, 1 if $x > y$, and 2 otherwise (denoted as $x?y$). If either operand is `nan`, stop the comparison with `__fp_compare_nan:w` returning 2 . If x is negative, swap the outputs 1 and -1 (i.e., $>$ and $<$); we can henceforth assume that $x \geq 0$. If $y \geq 0$, and they have the same type, either they are normal and we compare them with `__fp_compare_npos:nwnw`, or they

are equal. If $y \geq 0$, but of a different type, the highest type is a larger number. Finally, if $y \leq 0$, then $x > y$, unless both are zero.

```

13930 \cs_new:Npn \__fp_compare_back_any:ww #1#2; #3
13931 {
13932   \__fp_if_type_fp:NTwFw
13933   #1 { \__fp_if_type_fp:NTwFw #3 \use_i:nn \s__fp \use_ii:nn \q_stop }
13934   \s__fp \use_ii:nn \q_stop
13935   \__fp_compare_back:ww
13936   {
13937     \cs:w
13938     __fp
13939     \__fp_type_from_scan:N #1
13940     _compare_back
13941     \__fp_type_from_scan:N #3
13942     :ww
13943     \cs_end:
13944   }
13945   #1#2 ; #3
13946 }
13947 \cs_new:Npn \__fp_compare_back:ww
13948   \s__fp \__fp_chk:w #1 #2 #3;
13949   \s__fp \__fp_chk:w #4 #5 #6;
13950 {
13951   \int_value:w
13952   \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
13953   \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
13954   \if_meaning:w 2 #5 - \fi:
13955   \if_meaning:w #2 #5
13956   \if_meaning:w #1 #4
13957   \if_meaning:w 1 #1
13958   \__fp_compare_npos:nwnw #6; #3;
13959   \else:
13960     0
13961   \fi:
13962   \else:
13963     \if_int_compare:w #4 < #1 - \fi: 1
13964   \fi:
13965   \else:
13966     \if_int_compare:w #1#4 = 0 \exp_stop_f:
13967     0
13968   \else:
13969     1
13970   \fi:
13971   \fi:
13972   \exp_stop_f:
13973 }
13974 \cs_new:Npn \__fp_compare_nan:w #1 \fi: \exp_stop_f: { 2 \exp_stop_f: }
```

(End definition for __fp_compare_back_any:ww, __fp_compare_back:ww, and __fp_compare_nan:w.)

__fp_compare_back_tuple:ww Tuple and floating point numbers are not comparable so return 2 in mixed cases or
 __fp_tuple_compare_back:ww when tuples have a different number of items. Otherwise compare pairs of items with
 __fp_tuple_compare_back_tuple:ww __fp_compare_back_any:ww and if any don't match return 2 (as \int_value:w 02
 __fp_tuple_compare_back_loop:w \exp_stop_f:).

```

13975 \cs_new:Npn \__fp_compare_back_tuple:ww #1; #2; { 2 }
13976 \cs_new:Npn \__fp_tuple_compare_back:ww #1; #2; { 2 }
13977 \cs_new:Npn \__fp_tuple_compare_back_tuple:ww
13978   \s__fp_tuple \__fp_tuple_chk:w #1;
13979   \s__fp_tuple \__fp_tuple_chk:w #2;
13980   {
13981     \int_compare:nNnTF { \__fp_array_count:n {#1} } =
13982       { \__fp_array_count:n {#2} }
13983       {
13984         \int_value:w 0
13985         \__fp_tuple_compare_back_loop:w
13986           #1 { \s__fp \prg_break: } ; @
13987           #2 { \s__fp \prg_break: } ;
13988         \prg_break_point:
13989         \exp_stop_f:
13990       }
13991       { 2 }
13992   }
13993 \cs_new:Npn \__fp_tuple_compare_back_loop:w #1#2 ; #3 @ #4#5 ;
13994 {
13995   \use_none:n #1
13996   \use_none:n #4
13997   \if_int_compare:w
13998     \__fp_compare_back_any:ww #1 #2 ; #4 #5 ; = 0 \exp_stop_f:
13999   \else:
14000     2 \exp_after:wN \prg_break:
14001   \fi:
14002   \__fp_tuple_compare_back_loop:w #3 @
14003 }

```

(End definition for __fp_compare_back_tuple:ww and others.)

__fp_compare_npos:nwnw
 __fp_compare_significand:nnnnnnnn

__fp_compare_npos:nwnw {<expo₁>} <body₁> ; {<expo₂>} <body₂> ;
 Within an \int_value:w ... \exp_stop_f: construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```

14004 \cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
14005 {
14006   \if_int_compare:w #1 = #3 \exp_stop_f:
14007     \__fp_compare_significand:nnnnnnnn #2 #4
14008   \else:
14009     \if_int_compare:w #1 < #3 - \fi: 1
14010   \fi:
14011 }
14012 \cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
14013 {
14014   \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
14015   \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
14016     0
14017   \else:
14018     \if_int_compare:w #3#4 < #7#8 - \fi: 1

```

```

14019     \fi:
14020   \else:
14021     \if_int_compare:w #1#2 < #5#6 - \fi: 1
14022   \fi:
14023 }

```

(End definition for `__fp_compare_npos:nwnw` and `__fp_compare_significand:nnnnnnnn`.)

27.4 Floating point expression loops

`\fp_do_until:nn` These are quite easy given the above functions. The `do_until` and `do_while` versions execute the body, then test. The `until_do` and `while_do` do it the other way round.

```

14024 \cs_new:Npn \fp_do_until:nn #1#2
14025 {
14026   #2
14027   \fp_compare:nF {#1}
14028   { \fp_do_until:nn {#1} {#2} }
14029 }
14030 \cs_new:Npn \fp_do_while:nn #1#2
14031 {
14032   #2
14033   \fp_compare:nT {#1}
14034   { \fp_do_while:nn {#1} {#2} }
14035 }
14036 \cs_new:Npn \fp_until_do:nn #1#2
14037 {
14038   \fp_compare:nF {#1}
14039   {
14040     #2
14041     \fp_until_do:nn {#1} {#2}
14042   }
14043 }
14044 \cs_new:Npn \fp_while_do:nn #1#2
14045 {
14046   \fp_compare:nT {#1}
14047   {
14048     #2
14049     \fp_while_do:nn {#1} {#2}
14050   }
14051 }

```

(End definition for `\fp_do_until:nn` and others. These functions are documented on page 184.)

`\fp_do_until:nNnn` As above but not using the `nNn` syntax.

```

14052 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
14053 {
14054   #4
14055   \fp_compare:nNnF {#1} #2 {#3}
14056   { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
14057 }
14058 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
14059 {
14060   #4
14061   \fp_compare:nNnT {#1} #2 {#3}

```



```

14062     { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
14063   }
14064 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
14065 {
14066   \fp_compare:nNnF {#1} #2 {#3}
14067   {
14068     #4
14069     \fp_until_do:nNnn {#1} #2 {#3} {#4}
14070   }
14071 }
14072 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
14073 {
14074   \fp_compare:nNnT {#1} #2 {#3}
14075   {
14076     #4
14077     \fp_while_do:nNnn {#1} #2 {#3} {#4}
14078   }
14079 }

```

(End definition for `\fp_do_until:nNnn` and others. These functions are documented on page 183.)

\fp_step_function:nnnN

\fp_step_function:nnnc

`__fp_step:wwwN`

`__fp_step_fp:wwwN`

`__fp_step:NnnnnN`

`__fp_step:NfnnnN`

The approach here is somewhat similar to `\int_step_function:nnnN`. There are two subtleties: we use the internal parser `__fp_parse:n` to avoid converting back and forth from the internal representation; and (due to rounding) even a non-zero step does not guarantee that the loop counter increases.

```

14080 \cs_new:Npn \fp_step_function:nnnN #1#2#3
14081 {
14082   \exp_after:wN \__fp_step:wwwN
14083   \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#1}
14084   \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#2}
14085   \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
14086 }
14087 \cs_generate_variant:Nn \fp_step_function:nnnN { nnnc }
14088 % \end{macrocode}
14089 % Only floating point numbers (not tuples) are allowed arguments.
14090 % Only \enquote{normal} floating points (not $\pm 0$,
14091 % $\pm\texttt{inf}$, $\texttt{nan}$) can be used as step; if positive,
14092 % call \cs{__fp_step:NnnnnN} with argument |>| otherwise~|<|. This
14093 % function has one more argument than its integer counterpart, namely
14094 % the previous value, to catch the case where the loop has made no
14095 % progress. Conversion to decimal is done just before calling the
14096 % user's function.
14097 % \begin{macrocode}
14098 \cs_new:Npn \__fp_step:wwwN #1#2; #3#4; #5#6; #7
14099 {
14100   \__fp_if_type_fp:NTwFw #1 { } \s__fp \prg_break: \q_stop
14101   \__fp_if_type_fp:NTwFw #3 { } \s__fp \prg_break: \q_stop
14102   \__fp_if_type_fp:NTwFw #5 { } \s__fp \prg_break: \q_stop
14103   \use_i:nnnn { \__fp_step_fp:wwwN #1#2; #3#4; #5#6; #7 }
14104   \prg_break_point:
14105   \use:n
14106   {
14107     \__fp_error:nfff { fp-step-tuple } { \fp_to_tl:n { #1#2 ; } }
14108     { \fp_to_tl:n { #3#4 ; } } { \fp_to_tl:n { #5#6 ; } }

```

```

14109     }
14110   }
14111   \cs_new:Npn \__fp_step_fp:wwwN #1 ; \s__fp \__fp_chk:w #2#3#4 ; #5; #6
14112   {
14113     \token_if_eq_meaning:NNTF #2 1
14114     {
14115       \token_if_eq_meaning:NNTF #3 0
14116       { \__fp_step:NnnnnN > }
14117       { \__fp_step:NnnnnN < }
14118     }
14119     {
14120       \token_if_eq_meaning:NNTF #2 0
14121       {
14122         \__kernel_msg_expandable_error:nnn { kernel }
14123         { zero-step } {#6}
14124       }
14125       {
14126         \__fp_error:nnfn { fp-bad-step } { }
14127         { \fp_to_tl:n { \s__fp \__fp_chk:w #2#3#4 ; } } {#6}
14128       }
14129       \use_none:nnnnn
14130     }
14131     { #1 ; } { \c_nan_fp } { \s__fp \__fp_chk:w #2#3#4 ; } { #5 ; } #6
14132   }
14133   \cs_new:Npn \__fp_step:NnnnnN #1#2#3#4#5#6
14134   {
14135     \fp_compare:nNnTF {#2} = {#3}
14136     {
14137       \__fp_error:nffn { fp-tiny-step }
14138       { \fp_to_tl:n {#3} } { \fp_to_tl:n {#4} } {#6}
14139     }
14140     {
14141       \fp_compare:nNnF {#2} #1 {#5}
14142       {
14143         \exp_args:Nf #6 { \__fp_to_decimal_dispatch:w #2 }
14144         \__fp_step:NfnnnnN
14145         #1 { \__fp_parse:n { #2 + #4 } } {#2} {#4} {#5} #6
14146       }
14147     }
14148   }
14149   \cs_generate_variant:Nn \__fp_step:NnnnnN { Nf }

```

(End definition for \fp_step_function:nnnN and others. This function is documented on page 185.)

\fp_step_inline:nnnn As for \int_step_inline:nnnn, create a global function and apply it, following up with
\fp_step_variable:nnnNn a break point.

```

\__fp_step:NNnnnnn
14150 \cs_new_protected:Npn \fp_step_inline:nnnn
14151 {
14152   \int_gincr:N \g__kernel_prg_map_int
14153   \exp_args:NNc \__fp_step:NNnnnnn
14154   \cs_gset_protected:Npn
14155   { __fp_map_ \int_use:N \g__kernel_prg_map_int :w }
14156 }
14157 \cs_new_protected:Npn \fp_step_variable:nnnNn #1#2#3#4#5

```

```

14158 {
14159   \int_gincr:N \g__kernel_prg_map_int
14160   \exp_args:Nnc \__fp_step:NNnnnn
14161   \cs_gset_protected:Npx
14162     { \__fp_map_ \int_use:N \g__kernel_prg_map_int :w }
14163     {#1} {#2} {#3}
14164     {
14165       \tl_set:Nn \exp_not:N #4 {##1}
14166       \exp_not:n {#5}
14167     }
14168   }
14169   \cs_new_protected:Npn \__fp_step:NNnnnn #1#2#3#4#5#6
14170   {
14171     #1 #2 ##1 {#6}
14172     \fp_step_function:nnnN {#3} {#4} {#5} #2
14173     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
14174   }

```

(End definition for `\fp_step_inline:nnnn`, `\fp_step_variable:nnnN`, and `__fp_step:NNnnnn`. These functions are documented on page 185.)

```

14175 \__kernel_msg_new:nnn { kernel } { fp-step-tuple }
14176 { Tuple~argument~in~fp_step...~{#1}{#2}{#3}. }
14177 \__kernel_msg_new:nnn { kernel } { fp-bad-step }
14178 { Invalid~step~size~#2~in~step~function~#3. }
14179 \__kernel_msg_new:nnn { kernel } { fp-tiny-step }
14180 { Tiny~step~size~(#{1}+#{2}=#{1})~in~step~function~#3. }

```

27.5 Extrema

```

\__fp_minmax_o:Nw
\__fp_minmax_aux_o:Nw

```

First check all operands are floating point numbers. The argument #1 is 2 to find the maximum of an array #2 of floating point numbers, and 0 to find the minimum. We read numbers sequentially, keeping track of the largest (smallest) number found so far. If numbers are equal (for instance ± 0), the first is kept. We append $-\infty$ (∞), for the case of an empty array. Since no number is smaller (larger) than that, this additional item only affects the maximum (minimum) in the case of `max()` and `min()` with no argument. The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `__fp_minmax_loop:Nww`.

```

14181 \cs_new:Npn \__fp_minmax_o:Nw #1
14182 {
14183   \__fp_parse_function_all_fp_o:fnw
14184   { \token_if_eq_meaning:NNTF 0 #1 { min } { max } }
14185   { \__fp_minmax_aux_o:Nw #1 }
14186 }
14187 \cs_new:Npn \__fp_minmax_aux_o:Nw #1#2 @
14188 {
14189   \if_meaning:w 0 #1
14190     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN +
14191   \else:
14192     \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN -
14193   \fi:
14194   #2
14195   \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
14196   \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;

```

14197 }

(End definition for _fp_minmax_o:Nw and _fp_minmax_aux_o:Nw.)

_fp_minmax_loop:Nww

The first argument is $-$ or $+$ to denote the case where the currently largest (smallest) number found (first floating point argument) should be replaced by the new number (second floating point argument). If the new number is `nan`, keep that as the extremum, unless that extremum is already a `nan`. Otherwise, compare the two numbers. If the new number is larger (in the case of `max`) or smaller (in the case of `min`), the test yields `true`, and we keep the second number as a new maximum; otherwise we keep the first number. Then loop.

```

14198 \cs_new:Npn \_fp_minmax_loop:Nww
14199   #1 \s__fp \_fp_chk:w #2#3; \s__fp \_fp_chk:w #4#5;
14200   {
14201     \if_meaning:w 3 #4
14202     \if_meaning:w 3 #2
14203       \_fp_minmax_auxi:ww
14204     \else:
14205       \_fp_minmax_auxii:ww
14206     \fi:
14207   \else:
14208     \if_int_compare:w
14209       \_fp_compare_back:ww
14210       \s__fp \_fp_chk:w #4#5;
14211       \s__fp \_fp_chk:w #2#3;
14212       = #1 1 \exp_stop_f:
14213       \_fp_minmax_auxii:ww
14214     \else:
14215       \_fp_minmax_auxi:ww
14216     \fi:
14217   \fi:
14218   \_fp_minmax_loop:Nww #1
14219   \s__fp \_fp_chk:w #2#3;
14220   \s__fp \_fp_chk:w #4#5;
14221 }

```

(End definition for _fp_minmax_loop:Nww.)

_fp_minmax_auxi:ww

Keep the first/second number, and remove the other.

_fp_minmax_auxii:ww

```

14222 \cs_new:Npn \_fp_minmax_auxi:ww #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
14223   { \fi: \fi: #2 \s__fp #3 ; }
14224 \cs_new:Npn \_fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
14225   { \fi: \fi: #2 }

```

(End definition for _fp_minmax_auxi:ww and _fp_minmax_auxii:ww.)

_fp_minmax_break_o:w

This function is called from within an `\if_meaning:w` test. Skip to the end of the tests, close the current test with `\fi:`, clean up, and return the appropriate number with one post-expansion.

```

14226 \cs_new:Npn \_fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
14227   { \fi: \_fp_exp_after_o:w \s__fp #3; }

```

(End definition for _fp_minmax_break_o:w.)

27.6 Boolean operations

`__fp_not_o:w` Return true or false, with two expansions, one to exit the conditional, and one to please `l3fp-parse`. The first argument is provided by `l3fp-parse` and is ignored.

```

14228 \cs_new:Npn \__fp_not_o:w #1 \s__fp \__fp_chk:w #2#3; @
14229 {
14230   \if_meaning:w 0 #2
14231     \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
14232   \else:
14233     \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
14234   \fi:
14235 }
14236 \cs_new:Npn \__fp_tuple_not_o:w #1 @ { \exp_after:wN \c_zero_fp }
```

(End definition for `__fp_not_o:w` and `__fp_tuple_not_o:w`.)

`__fp_&_o:ww` For `and`, if the first number is zero, return it (with the same sign). Otherwise, return the second one. For `or`, the logic is reversed: if the first number is non-zero, return it, otherwise return the second number: we achieve that by hi-jacking `__fp_&_o:ww`, inserting an extra argument, `\else:`, before `\s__fp`. In all cases, expand after the floating point number.

```

14237 \group_begin:
14238   \char_set_catcode_letter:N &
14239   \char_set_catcode_letter:N |
14240   \cs_new:Npn \__fp_&_o:ww #1 \s__fp \__fp_chk:w #2#3;
14241   {
14242     \if_meaning:w 0 #2 #1
14243       \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
14244     \fi:
14245     \__fp_exp_after_o:w
14246   }
14247   \cs_new:Npn \__fp_&_tuple_o:ww #1 \s__fp \__fp_chk:w #2#3;
14248   {
14249     \if_meaning:w 0 #2 #1
14250       \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
14251     \fi:
14252     \__fp_exp_after_tuple_o:w
14253   }
14254   \cs_new:Npn \__fp_tuple_&_o:ww #1; { \__fp_exp_after_o:w }
14255   \cs_new:Npn \__fp_tuple_&_tuple_o:ww #1; { \__fp_exp_after_tuple_o:w }
14256   \cs_new:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }
14257   \cs_new:Npn \__fp_|_tuple_o:ww { \__fp_&_tuple_o:ww \else: }
14258   \cs_new:Npn \__fp_tuple_|_o:ww #1; #2; { \__fp_exp_after_tuple_o:w #1; }
14259   \cs_new:Npn \__fp_tuple_|_tuple_o:ww #1; #2;
14260     { \__fp_exp_after_tuple_o:w #1; }
14261 \group_end:
14262 \cs_new:Npn \__fp_and_return:wNw #1; \fi: #2;
14263 { \fi: \__fp_exp_after_o:w #1; }
```

(End definition for `__fp_&_o:ww` and others.)

27.7 Ternary operator

`__fp_ternary:NwN`
`__fp_ternary_auxi:NwN`
`__fp_ternary_auxii:NwN`

The first function receives the test and the true branch of the `?:` ternary operator. It calls `__fp_ternary_auxii:NwN` if the test branch is a floating point number ± 0 , and otherwise calls `__fp_ternary_auxi:NwN`. These functions select one of their two arguments.

```

14264 \cs_new:Npn \__fp_ternary:NwN #1 #2#3@ #4@ #5
14265 {
14266   \if_meaning:w \__fp_parse_infix_:N #5
14267   \if_charcode:w 0
14268     \__fp_if_type_fp:NTwFw
14269     #2 { \use_i:nn \use_i_delimit_by_q_stop:nw #3 \q_stop }
14270     \s__fp 1 \q_stop
14271     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxii:NwN
14272   \else:
14273     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxi:NwN
14274   \fi:
14275   \exp_after:wN #1
14276   \exp:w \exp_end_continue_f:w
14277   \__fp_exp_after_array_f:w #4 \s__fp_stop
14278   \exp_after:wN @
14279   \exp:w
14280     \__fp_parse_operand:Nw \c__fp_prec_colon_int
14281     \__fp_parse_expand:w
14282   \else:
14283     \__kernel_msg_expandable_error:nnnn
14284     { kernel } { fp-missing } { : } { ~for~?: }
14285     \exp_after:wN \__fp_parse_continue:NwN
14286     \exp_after:wN #1
14287     \exp:w \exp_end_continue_f:w
14288     \__fp_exp_after_array_f:w #4 \s__fp_stop
14289     \exp_after:wN #5
14290     \exp_after:wN #1
14291   \fi:
14292 }
14293 \cs_new:Npn \__fp_ternary_auxi:NwN #1#2@#3@#4
14294 {
14295   \exp_after:wN \__fp_parse_continue:NwN
14296   \exp_after:wN #1
14297   \exp:w \exp_end_continue_f:w
14298   \__fp_exp_after_array_f:w #2 \s__fp_stop
14299   #4 #1
14300 }
14301 \cs_new:Npn \__fp_ternary_auxii:NwN #1#2@#3@#4
14302 {
14303   \exp_after:wN \__fp_parse_continue:NwN
14304   \exp_after:wN #1
14305   \exp:w \exp_end_continue_f:w
14306   \__fp_exp_after_array_f:w #3 \s__fp_stop
14307   #4 #1
14308 }

```

(End definition for `__fp_ternary:NwN`, `__fp_ternary_auxi:NwN`, and `__fp_ternary_auxii:NwN`.)

14309 `/\initex | package)`

28 l3fp-basics Implementation

14310 $\langle *initex | package \rangle$

14311 $\langle @@=fp \rangle$

The `l3fp-basics` module implements addition, subtraction, multiplication, and division of two floating points, and the absolute value and sign-changing operations on one floating point. All operations implemented in this module yield the outcome of rounding the infinitely precise result of the operation to the nearest floating point.

Some algorithms used below end up being quite similar to some described in “What Every Computer Scientist Should Know About Floating Point Arithmetic”, by David Goldberg, which can be found at <http://cr.yp.to/2005-590/goldberg.pdf>.

Unary functions.

`__fp_parse_word_abs:N`
`__fp_parse_word_sign:N`
`__fp_parse_word_sqrt:N`

```
14312 \cs_new:Npn __fp_parse_word_abs:N
14313   { __fp_parse_unary_function:NNN __fp_set_sign_o:w 0 }
14314 \cs_new:Npn __fp_parse_word_sign:N
14315   { __fp_parse_unary_function:NNN __fp_sign_o:w ? }
14316 \cs_new:Npn __fp_parse_word_sqrt:N
14317   { __fp_parse_unary_function:NNN __fp_sqrt_o:w ? }
```

(End definition for `__fp_parse_word_abs:N`, `__fp_parse_word_sign:N`, and `__fp_parse_word_sqrt:N`.)

28.1 Addition and subtraction

We define here two functions, `__fp-_o:ww` and `__fp+_o:ww`, which perform the subtraction and addition of their two floating point operands, and expand the tokens following the result once.

A more obscure function, `__fp_add_big_i_o:wNww`, is used in `l3fp-expo`.

The logic goes as follows:

- `__fp-_o:ww` calls `__fp+_o:ww` to do the work, with the sign of the second operand flipped;
- `__fp+_o:ww` dispatches depending on the type of floating point, calling specialized auxiliaries;
- in all cases except summing two normal floating point numbers, we return one or the other operands depending on the signs, or detect an invalid operation in the case of $\infty - \infty$;
- for normal floating point numbers, compare the signs;
- to add two floating point numbers of the same sign or of opposite signs, shift the significand of the smaller one to match the bigger one, perform the addition or subtraction of significands, check for a carry, round, and pack using the `__fp-basics_pack_...` functions.

The trickiest part is to round correctly when adding or subtracting normal floating point numbers.

28.1.1 Sign, exponent, and special numbers

`__fp_-_o:ww` The `__fp+_o:ww` auxiliary has a hook: it takes one argument between the first `\s__fp` and `__fp_chk:w`, which is applied to the sign of the second operand. Positioning the hook there means that `__fp+_o:ww` can still perform the sanity check that it was followed by `\s__fp`.

```

14318 \cs_new:cpx { __fp_-_o:ww } \s__fp
14319 {
14320     \exp_not:c { __fp+_o:ww }
14321     \exp_not:n { \s__fp \__fp_neg_sign:N }
14322 }
```

(End definition for `__fp_-_o:ww`.)

`__fp+_o:ww` This function is either called directly with an empty #1 to compute an addition, or it is called by `__fp_-_o:ww` with `__fp_neg_sign:N` as #1 to compute a subtraction, in which case the second operand's sign should be changed. If the $\langle types \rangle$ #2 and #4 are the same, dispatch to case #2 (0, 1, 2, or 3), where we call specialized functions: thanks to `\int_value:w`, those receive the tweaked $\langle sign_2 \rangle$ (expansion of #1#5) as an argument. If the $\langle types \rangle$ are distinct, the result is simply the floating point number with the highest $\langle type \rangle$. Since case 3 (used for two nan) also picks the first operand, we can also use it when $\langle type_1 \rangle$ is greater than $\langle type_2 \rangle$. Also note that we don't need to worry about $\langle sign_2 \rangle$ in that case since the second operand is discarded.

```

14323 \cs_new:cpn { __fp+_o:ww }
14324     \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
14325 {
14326     \if_case:w
14327         \if_meaning:w #2 #4
14328             #2
14329         \else:
14330             \if_int_compare:w #2 > #4 \exp_stop_f:
14331                 3
14332             \else:
14333                 4
14334             \fi:
14335         \fi:
14336     \exp_stop_f:
14337         \exp_after:wN \__fp_add_zeros_o:Nww \int_value:w
14338     \or: \exp_after:wN \__fp_add_normal_o:Nww \int_value:w
14339     \or: \exp_after:wN \__fp_add_inf_o:Nww \int_value:w
14340     \or: \__fp_case_return_i_o:ww
14341     \else: \exp_after:wN \__fp_add_return_ii_o:Nww \int_value:w
14342     \fi:
14343     #1 #5
14344     \s__fp \__fp_chk:w #2 #3 ;
14345     \s__fp \__fp_chk:w #4 #5
14346 }
```

(End definition for `__fp+_o:ww`.)

`__fp_add_return_ii_o:Nww` Ignore the first operand, and return the second, but using the sign #1 rather than #4. As usual, expand after the floating point.

```

14347 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
14348 { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }
```


(End definition for _fp_add_return_ii_o:Nww.)

_fp_add_zeros_o:Nww Adding two zeros yields \c_zero_fp, except if both zeros were -0 .

```

14349 \cs_new:Npn \_fp_add_zeros_o:Nww #1 \s__fp \_fp_chk:w 0 #2
14350 {
14351   \if_int_compare:w #2 #1 = 20 \exp_stop_f:
14352     \exp_after:wN \_fp_add_return_ii_o:Nww
14353   \else:
14354     \_fp_case_return_i_o:ww
14355   \fi:
14356   #1
14357   \s__fp \_fp_chk:w 0 #2
14358 }

```

(End definition for _fp_add_zeros_o:Nww.)

_fp_add_inf_o:Nww If both infinities have the same sign, just return that infinity, otherwise, it is an invalid operation. We find out if that invalid operation is an addition or a subtraction by testing whether the tweaked $\langle sign_2 \rangle$ (#1) and the $\langle sign_2 \rangle$ (#4) are identical.

```

14359 \cs_new:Npn \_fp_add_inf_o:Nww
14360   #1 \s__fp \_fp_chk:w 2 #2 #3; \s__fp \_fp_chk:w 2 #4
14361 {
14362   \if_meaning:w #1 #2
14363     \_fp_case_return_i_o:ww
14364   \else:
14365     \_fp_case_use:nw
14366     {
14367       \exp_last_unbraced:Nf \_fp_invalid_operation_o:Nww
14368       { \token_if_eq_meaning:NNTF #1 #4 + - }
14369     }
14370   \fi:
14371   \s__fp \_fp_chk:w 2 #2 #3;
14372   \s__fp \_fp_chk:w 2 #4
14373 }

```

(End definition for _fp_add_inf_o:Nww.)

_fp_add_normal_o:Nww _fp_add_normal_o:Nww $\langle sign_2 \rangle$ \s__fp _fp_chk:w 1 $\langle sign_1 \rangle$ $\langle exp_1 \rangle$ $\langle body_1 \rangle$; \s__fp _fp_chk:w 1 $\langle initial\ sign_2 \rangle$ $\langle exp_2 \rangle$ $\langle body_2 \rangle$;

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```

14374 \cs_new:Npn \_fp_add_normal_o:Nww #1 \s__fp \_fp_chk:w 1 #2
14375 {
14376   \if_meaning:w #1#2
14377     \exp_after:wN \_fp_add_npos_o:NnwNnw
14378   \else:
14379     \exp_after:wN \_fp_sub_npos_o:NnwNnw
14380   \fi:
14381   #2
14382 }

```

(End definition for _fp_add_normal_o:Nww.)

28.1.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

$$\backslash_fp_add_npos_o:NnwNnw \langle sign_1 \rangle \langle exp_1 \rangle \langle body_1 \rangle ; \backslash s_fp \backslash_fp_chk:w 1$$

Since we are doing an addition, the final sign is $\langle sign_1 \rangle$. Start an `_fp_int_eval:w`, responsible for computing the exponent: the result, and the $\langle final\ sign \rangle$ are then given to `_fp_sanitize:Nw` which checks for overflow. The exponent is computed as the largest exponent #2 or #5, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by `_fp_add_big_i:wNww` or `_fp_add_big_ii:wNww`. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```

14383 \cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
14384 {
14385   \exp_after:wN \__fp_sanitizewN
14386   \exp_after:wN #1
14387   \int_value:w \__fp_int_eval:w
14388   \if_int_compare:w #2 > #5 \exp_stop_f:
14389     #2
14390     \exp_after:wN \__fp_add_big_i_o:wNnw \int_value:w -
14391   \else:
14392     #5
14393     \exp_after:wN \__fp_add_big_ii_o:wNnw \int_value:w
14394   \fi:
14395   \__fp_int_eval:w #5 - #2 ; #1 #3;
14396 }

```

(End definition for _fp_add_npos_o:NnwNnw.)

$$\backslash_fp_add_big_i_o:wNww \langle shift \rangle ; \langle final\ sign \rangle \langle body_1 \rangle ; \langle body_2 \rangle ;$$

```

\__fp_add_big_ii_o:wNww      Used in l3fp-exo. Shift the significand of the small number, then add with \__fp_
add significand o:NnnwnnnnnN.

```

```

14397 \cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;
14398 {
14399     \__fp_decimate:nNnnnn {#1}
14400     \__fp_add_significand_o:NnnwnnnnnN
14401     #4
14402     #3
14403     #2
14404 }
14405 \cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
14406 {
14407     \__fp_decimate:nNnnnn {#1}
14408     \__fp_add_significand_o:NnnwnnnnnN
14409     #3
14410     #4
14411     #2
14412 }

```

(End definition for `_fp_add_big_i_o:wNww` and `_fp_add_big_ii_o:wNww`.)

```

\__fp_add_significand_o:NnnwnnnnN \__fp_add_significand_o:NnnwnnnnN <rounding digit> {\langle Y'_1 \rangle} {\langle Y'_2 \rangle}
\__fp_add_significand_pack:NNNNNNN <extra-digits> ; {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} <final sign>
\__fp_add_significand_test_o:N

```

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as $0.99\dots95 \rightarrow 1.00\dots0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```

14413 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
14414 {
14415   \exp_after:wN \__fp_add_significand_test_o:N
14416   \int_value:w \__fp_int_eval:w 1#5#6 + #2
14417   \exp_after:wN \__fp_add_significand_pack:NNNNNNN
14418   \int_value:w \__fp_int_eval:w 1#7#8 + #3 ; #1
14419 }
14420 \cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
14421 {
14422   \if_meaning:w 2 #1
14423     + 1
14424   \fi:
14425   ; #2 #3 #4 #5 #6 #7 ;
14426 }
14427 \cs_new:Npn \__fp_add_significand_test_o:N #1
14428 {
14429   \if_meaning:w 2 #1
14430     \exp_after:wN \__fp_add_significand_carry_o:wwwNN
14431   \else:
14432     \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
14433   \fi:
14434 }

```

(End definition for `__fp_add_significand_o:NnnwnnnnN`, `__fp_add_significand_pack:NNNNNNN`, and `__fp_add_significand_test_o:N`.)

```

\__fp_add_significand_no_carry_o:wwwNN \__fp_add_significand_no_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>

```

If there's no carry, grab all the digits again and round. The packing function `__fp_basics_pack_high:NNNNNw` takes care of the case where rounding brings a carry.

```

14435 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
14436   #1; #2; #3#4 ; #5#6
14437 {
14438   \exp_after:wN \__fp_basics_pack_high:NNNNNw
14439   \int_value:w \__fp_int_eval:w 1 #1
14440   \exp_after:wN \__fp_basics_pack_low:NNNNNw
14441   \int_value:w \__fp_int_eval:w 1 #2 #3#4
14442   + \__fp_round:NNN #6 #4 #5
14443   \exp_after:wN ;
14444 }

```

(End definition for `__fp_add_significand_no_carry_o:wwwNN`.)

```

\__fp_add_significand_carry_o:wwwNN \__fp_add_significand_carry_o:wwwNN <8d> ; <6d> ; <2d> ; <rounding
digit> <sign>

```

The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don't care.

```

14445 \cs_new:Npn \__fp_add_significand_carry_o:wwwNN
14446   #1; #2; #3#4; #5#6
14447   {
14448     + 1
14449     \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
14450     \int_value:w \__fp_int_eval:w 1 1 #1
14451     \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
14452     \int_value:w \__fp_int_eval:w 1 #2#3 +
14453     \exp_after:wN \__fp_round:NNN
14454     \exp_after:wN #6
14455     \exp_after:wN #3
14456     \int_value:w \__fp_round_digit:Nw #4 #5 ;
14457     \exp_after:wN ;
14458   }

```

(End definition for __fp_add_significand_carry_o:wwwNN.)

28.1.3 Absolute subtraction

```

\__fp_sub_npos_o:NnwNnw \__fp_sub_npos_o:NnwNnw <sign1> <exp1> <body1> ; \s__fp \__fp_chk:w 1
\__fp_sub_eq_o:Nwnnw <initial sign2> <exp2> <body2> ;
\__fp_sub_npos_ii_o:Nwnnw

```

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call __fp_sub_npos_i_o:Nwnnw with the opposite of $\langle sign_1 \rangle$.

```

14459 \cs_new:Npn \__fp_sub_npos_o:NnwNnw #1#2#3; \s__fp \__fp_chk:w 1 #4#5#6;
14460   {
14461     \if_case:w \__fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
14462     \exp_after:wN \__fp_sub_eq_o:Nwnnw
14463     \or:
14464     \exp_after:wN \__fp_sub_npos_i_o:Nwnnw
14465     \else:
14466     \exp_after:wN \__fp_sub_npos_ii_o:Nwnnw
14467     \fi:
14468     #1 {#2} #3; {#5} #6;
14469   }
14470 \cs_new:Npn \__fp_sub_eq_o:Nwnnw #1#2; #3; { \exp_after:wN \c_zero_fp }
14471 \cs_new:Npn \__fp_sub_npos_ii_o:Nwnnw #1 #2; #3;
14472   {
14473     \exp_after:wN \__fp_sub_npos_i_o:Nwnnw
14474     \int_value:w \__fp_neg_sign:N #1
14475     #3; #2;
14476   }

```

(End definition for __fp_sub_npos_o:NnwNnw, __fp_sub_eq_o:Nwnnw, and __fp_sub_npos_ii_o:Nwnnw.)

__fp_sub_npos_i_o:Nwnnw After the computation is done, __fp_sanitize:Nw checks for overflow/underflow. It expects the $\langle final\ sign \rangle$ and the $\langle exponent \rangle$ (delimited by ;). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the `near` auxiliary. Otherwise, decimate y , then call the `far` auxiliary to evaluate

the difference between the two significands. Note that we decimate by 1 less than one could expect.

```

14477 \cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
14478 {
14479   \exp_after:wN \__fp_sanitizize:Nw
14480   \exp_after:wN #1
14481   \int_value:w \__fp_int_eval:w
14482   #2
14483   \if_int_compare:w #2 = #4 \exp_stop_f:
14484     \exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
14485   \else:
14486     \exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN
14487     { \int_value:w \__fp_int_eval:w #2 - #4 - 1 \exp_after:wN }
14488     \exp_after:wN \__fp_sub_back_far_o:NnnwnnnnnN
14489   \fi:
14490   #5
14491   #3
14492   #1
14493 }

```

(End definition for __fp_sub_npos_i_o:Nnwnw.)

```

\__fp_sub_back_near_o:nnnnnnnnN    \__fp_sub_back_near_o:nnnnnnnnN {<Y1>} {<Y2>} {<Y3>} {<Y4>} {<X1>}
\__fp_sub_back_near_pack:NNNNNNw   {<X2>} {<X3>} {<X4>} {<final sign>}
\__fp_sub_back_near_after:wNNNNw

```

In this case, the subtraction is exact, so we discard the *<final sign>* #9. The very large shifts of 10^9 and $1.1 \cdot 10^9$ are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

```

14494 \cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
14495 {
14496   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
14497   \int_value:w \__fp_int_eval:w 10#5#6 - #1#2 - 11
14498   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
14499   \int_value:w \__fp_int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
14500 }
14501 \cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
14502 { + #1#2 ; {#3#4#5#6} {#7} ; }
14503 \cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
14504 {
14505   \if_meaning:w 0 #1
14506     \exp_after:wN \__fp_sub_back_shift:wnnnn
14507   \fi:
14508   ; {#1#2#3#4} {#5}
14509 }

```

(End definition for __fp_sub_back_near_o:nnnnnnnnN, __fp_sub_back_near_pack:NNNNNNw, and __fp_sub_back_near_after:wNNNNw.)

```

\__fp_sub_back_shift:wnnnn    \__fp_sub_back_shift:wnnnn ; {<Z1>} {<Z2>} {<Z3>} {<Z4>} ;
\__fp_sub_back_shift_ii:ww    This function is called with <Z1> ≤ 999. Act with \number to trim leading zeros from
\__fp_sub_back_shift_iii:NNNNNNw {<Z1> <Z2> (we don't do all four blocks at once, since non-zero blocks would then overflow
\__fp_sub_back_shift_iv:nnnnw  TEX's integers). If the first two blocks are zero, the auxiliary receives an empty #1 and

```

trims #2#30 from leading zeros, yielding a total shift between 7 and 16 to the exponent. Otherwise we get the shift from #1 alone, yielding a result between 1 and 6. Once the

exponent is taken care of, trim leading zeros from #1#2#3 (when #1 is empty, the space before #2#3 is ignored), get four blocks of 4 digits and finally clean up. Trailing zeros are added so that digits can be grabbed safely.

```

14510 \cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
14511 {
14512   \exp_after:wN \__fp_sub_back_shift_ii:ww
14513   \int_value:w #1 #2 0 ;
14514 }
14515 \cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
14516 {
14517   \if_meaning:w @ #1 @
14518   - 7
14519   - \exp_after:wN \use_i:nnn
14520     \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
14521     \int_value:w #2#3 0 ~ 123456789;
14522   \else:
14523     - \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
14524   \fi:
14525   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
14526   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
14527   \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
14528   \exp_after:wN ;
14529   \int_value:w
14530   #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
14531 }
14532 \cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
14533 \cs_new:Npn \__fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }

```

(End definition for __fp_sub_back_shift:wnnnn and others.)

__fp_sub_back_far_o:NnnwnnnnN $\langle \text{rounding} \rangle \{ \langle Y'_1 \rangle \} \{ \langle Y'_2 \rangle \}$
 $\langle \text{extra-digits} \rangle ; \{ \langle X_1 \rangle \} \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \} \langle \text{final sign} \rangle$

If the difference is greater than $10^{\langle expo_x \rangle}$, call the `very_far` auxiliary. If the result is less than $10^{\langle expo_x \rangle}$, call the `not_far` auxiliary. If it is too close a call to know yet, namely if $1 \langle Y'_1 \rangle \langle Y'_2 \rangle = \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle 0$, then call the `quite_far` auxiliary. We use the odd combination of space and semi-colon delimiters to allow the `not_far` auxiliary to grab each piece individually, the `very_far` auxiliary to use `__fp_pack_eight:wNNNNNNNN`, and the `quite_far` to ignore the significands easily (using the `; delimiter`).

```

14534 \cs_new:Npn \__fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
14535 {
14536   \if_case:w
14537     \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
14538     \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
14539     0
14540   \else:
14541     \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: 1
14542   \fi:
14543   \else:
14544     \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: 1
14545   \fi:
14546   \exp_stop_f:
14547     \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
14548   \or: \exp_after:wN \__fp_sub_back_very_far_o:wwwNN

```

```

14549     \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwNNN
14550     \fi:
14551     #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
14552 }

```

(End definition for __fp_sub_back_far_o:NnnwnnnnN.)

__fp_sub_back_quite_far_o:wwNN
__fp_sub_back_quite_far_ii:NN

The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit of x is 1, and all others vanish when subtracting y . Then the *rounding* #3 and the *final sign* #4 control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we get 1 whenever the *rounding* digit is less than or equal to 5 (remember that the *rounding* digit is only equal to 5 if there was no further non-zero digit).

```

14553 \cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
14554 {
14555     \exp_after:wN \__fp_sub_back_quite_far_ii:NN
14556     \exp_after:wN #3
14557     \exp_after:wN #4
14558 }
14559 \cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
14560 {
14561     \if_case:w \__fp_round_neg:NNN #2 0 #1
14562     \exp_after:wN \use_i:nn
14563     \else:
14564     \exp_after:wN \use_ii:nn
14565     \fi:
14566     { ; {1000} {0000} {0000} {0000} ; }
14567     { - 1 ; {9999} {9999} {9999} {9999} ; }
14568 }

```

(End definition for __fp_sub_back_quite_far_o:wwNN and __fp_sub_back_quite_far_ii:NN.)

__fp_sub_back_not_far_o:wwwNN

In the present case, x and y have different exponents, but y is large enough that $x - y$ has a smaller exponent than x . Decrement the exponent (with -1). Then proceed in a way similar to the *near* auxiliaries seen earlier, but multiplying x by 10 (#30 and #40 below), and with the added quirk that the *rounding* digit has to be taken into account. Namely, we may have to decrease the result by one unit if __fp_round_neg:NNN returns 1. This function expects the *final sign* #6, the last digit of 1100000000+#40-#2, and the *rounding* digit. Instead of redoing the computation for the second argument, we note that __fp_round_neg:NNN only cares about its parity, which is identical to that of the last digit of #2.

```

14569 \cs_new:Npn \__fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
14570 {
14571     - 1
14572     \exp_after:wN \__fp_sub_back_near_after:wNNNNw
14573     \int_value:w \__fp_int_eval:w 1#30 - #1 - 11
14574     \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
14575     \int_value:w \__fp_int_eval:w 11 0000 0000 + #40 - #2
14576     - \exp_after:wN \__fp_round_neg:NNN
14577     \exp_after:wN #6
14578     \use_none:nnnnnnn #2 #5
14579     \exp_after:wN ;
14580 }

```

(End definition for __fp_sub_back_not_far_o:wwwNN.)

_fp_sub_back_very_far_o:wwwNN
_fp_sub_back_very_far_ii_o:nnNwwNN

The case where $x - y$ and x have the same exponent is a bit more tricky, mostly because it cannot reuse the same auxiliaries. Shift the y significand by adding a leading 0. Then the logic is similar to the `not_far` functions above. Rounding is a bit more complicated: we have two *rounding* digits #3 and #6 (from the decimation, and from the new shift) to take into account, and getting the parity of the main result requires a computation. The first `\int_value:w` triggers the second one because the number is unfinished; we can thus not use 0 in place of 2 there.

```

14581 \cs_new:Npn \__fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
14582 {
14583   \__fp_pack_eight:wNNNNNNNN
14584   \__fp_sub_back_very_far_ii_o:nnNwwNN
14585   { 0 #1#2#3 #4#5#6#7 }
14586   ;
14587 }
14588 \cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
14589 {
14590   \exp_after:wN \__fp_basics_pack_high:NNNNw
14591   \int_value:w \__fp_int_eval:w 1#4 - #1 - 1
14592   \exp_after:wN \__fp_basics_pack_low:NNNNw
14593   \int_value:w \__fp_int_eval:w 2#5 - #2
14594   - \exp_after:wN \__fp_round_neg:NNN
14595   \exp_after:wN #7
14596   \int_value:w
14597   \if_int_odd:w \__fp_int_eval:w #5 - #2 \__fp_int_eval_end:
14598   1 \else: 2 \fi:
14599   \int_value:w \__fp_round_digit:Nw #3 #6 ;
14600   \exp_after:wN ;
14601 }

```

(End definition for `__fp_sub_back_very_far_o:wwwNN` and `__fp_sub_back_very_far_ii_o:nnNwwNN`.)

28.2 Multiplication

28.2.1 Signs, and special numbers

_fp*_o:ww

We go through an auxiliary, which is common with `_fp/_o:ww`. The first argument is the operation, used for the invalid operation exception. The second is inserted in a formula to dispatch cases slightly differently between multiplication and division. The third is the operation for normal floating points. The fourth is there for extra cases needed in `_fp/_o:ww`.

```

14602 \cs_new:cpn { \__fp*_o:ww }
14603 {
14604   \__fp_mul_cases_o:NnNww
14605   *
14606   { - 2 + }
14607   \__fp_mul_npos_o:Nww
14608   { }
14609 }

```

(End definition for `_fp*_o:ww`.)

_fp_mul_cases_o:nNnnww

Split into 10 cases (12 for division). If both numbers are normal, go to case 0 (same sign) or case 1 (opposite signs): in both cases, call `__fp_mul_npos_o:Nww` to do the work. If

the first operand is `nan`, go to case 2, in which the second operand is discarded; if the second operand is `nan`, go to case 3, in which the first operand is discarded (note the weird interaction with the final test on signs). Then we separate the case where the first number is normal and the second is zero: this goes to cases 4 and 5 for multiplication, 10 and 11 for division. Otherwise, we do a computation which dispatches the products $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$ to case 4 or 5 depending on the combined sign, the products $0 \times \infty$ and $\infty \times 0$ to case 6 or 7 (invalid operation), and the products $1 \times \infty = \infty \times 1 = \infty \times \infty = \infty$ to cases 8 and 9. Note that the code for these two cases (which return $\pm\infty$) is inserted as argument #4, because it differs in the case of divisions.

```

14610 \cs_new:Npn \__fp_mul_cases_o:NnNnw
14611   #1#2#3#4 \s__fp \__fp_chk:w #5#6#7; \s__fp \__fp_chk:w #8#9
14612 {
14613   \if_case:w \__fp_int_eval:w
14614     \if_int_compare:w #5 #8 = 11 ~
14615     1
14616   \else:
14617     \if_meaning:w 3 #8
14618     3
14619   \else:
14620     \if_meaning:w 3 #5
14621     2
14622   \else:
14623     \if_int_compare:w #5 #8 = 10 ~
14624     9 #2 - 2
14625   \else:
14626     (#5 #2 #8) / 2 * 2 + 7
14627   \fi:
14628   \fi:
14629   \fi:
14630   \fi:
14631   \if_meaning:w #6 #9 - 1 \fi:
14632   \__fp_int_eval_end:
14633   \__fp_case_use:nw { #3 0 }
14634   \or: \__fp_case_use:nw { #3 2 }
14635   \or: \__fp_case_return_i_o:ww
14636   \or: \__fp_case_return_ii_o:ww
14637   \or: \__fp_case_return_o:Nww \c_zero_fp
14638   \or: \__fp_case_return_o:Nww \c_minus_zero_fp
14639   \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
14640   \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
14641   \or: \__fp_case_return_o:Nww \c_inf_fp
14642   \or: \__fp_case_return_o:Nww \c_minus_inf_fp
14643   #4
14644   \fi:
14645   \s__fp \__fp_chk:w #5 #6 #7;
14646   \s__fp \__fp_chk:w #8 #9
14647 }

```

(End definition for `__fp_mul_cases_o:nNnnnw`.)

28.2.2 Absolute multiplication

In this subsection, we perform the multiplication of two positive normal numbers.

```

\__fp_mul_npos_o:Nww \__fp_mul_npos_o:Nww <final sign> \s__fp \__fp_chk:w 1 <sign1> {<exp1>}
<body1> ; \s__fp \__fp_chk:w 1 <sign2> {<exp2>} <body2> ;

```

After the computation, `__fp_sanitize:Nw` checks for overflow or underflow. As we did for addition, `__fp_int_eval:w` computes the exponent, catching any shift coming from the computation in the significand. The *<final sign>* is needed to do the rounding properly in the significand computation. We setup the post-expansion here, triggered by `__fp_mul_significand_o:nnnnNnnnn`.

This is also used in `l3fp-convert`.

```

14648 \cs_new:Npn \__fp_mul_npos_o:Nww
14649 #1 \s__fp \__fp_chk:w #2 #3 #4 #5 ; \s__fp \__fp_chk:w #6 #7 #8 #9 ;
14650 {
14651   \exp_after:wN \__fp_sanitize:Nw
14652   \exp_after:wN #1
14653   \int_value:w \__fp_int_eval:w
14654     #4 + #8
14655   \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
14656 }

```

(End definition for `__fp_mul_npos_o:Nww`.)

```

\__fp_mul_significand_o:nnnnNnnnn \__fp_mul_significand_o:nnnnNnnnn {<X1>} {<X2>} {<X3>} {<X4>} <sign>
\__fp_mul_significand_drop:NNNNNw {<Y1>} {<Y2>} {<Y3>} {<Y4>}
\__fp_mul_significand_keep:NNNNNw

```

Note the three semicolons at the end of the definition. One is for the last `__fp_mul_significand_drop:NNNNNw`; one is for `__fp_round_digit:Nw` later on; and one, preceded by `\exp_after:wN`, which is correctly expanded (within an `__fp_int_eval:w`), is used by `__fp_basics_pack_low:NNNNNw`.

The product of two 16 digit integers has 31 or 32 digits, but it is impossible to know which one before computing. The place where we round depends on that number of digits, and may depend on all digits until the last in some rare cases. The approach is thus to compute the 5 first blocks of 4 digits (the first one is between 100 and 9999 inclusive), and a compact version of the remaining 3 blocks. Afterwards, the number of digits is known, and we can do the rounding within yet another set of `__fp_int_eval:w`.

```

14657 \cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
14658 {
14659   \exp_after:wN \__fp_mul_significand_test_f:NNN
14660   \exp_after:wN #5
14661   \int_value:w \__fp_int_eval:w 99990000 + #1*#6 +
14662   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
14663   \int_value:w \__fp_int_eval:w 99990000 + #1*#7 + #2*#6 +
14664   \exp_after:wN \__fp_mul_significand_keep:NNNNNw
14665   \int_value:w \__fp_int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
14666   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
14667   \int_value:w \__fp_int_eval:w 99990000 + #1*#9 + #2*#8 +
14668   #3*#7 + #4*#6 +
14669   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
14670   \int_value:w \__fp_int_eval:w 99990000 + #2*#9 + #3*#8 +
14671   #4*#7 +
14672   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
14673   \int_value:w \__fp_int_eval:w 99990000 + #3*#9 + #4*#8 +
14674   \exp_after:wN \__fp_mul_significand_drop:NNNNNw
14675   \int_value:w \__fp_int_eval:w 100000000 + #4*#9 ;
14676 ; \exp_after:wN ;

```

```

14677 }
14678 \cs_new:Npn \__fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
14679 { #1#2#3#4#5 ; + #6 }
14680 \cs_new:Npn \__fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
14681 { #1#2#3#4#5 ; #6 ; }

```

(End definition for __fp_mul_significand_o:nnnnNnnnn, __fp_mul_significand_drop:NNNNNw, and __fp_mul_significand_keep:NNNNNw.)

```

\__fp_mul_significand_test_f:NNN \__fp_mul_significand_test_f:NNN <sign> 1 <digits 1-8> ; <digits 9-12> ;
<digits 13-16> ; + <digits 17-20> + <digits 21-24> + <digits 25-28> + <digits
29-32> ; \exp_after:wN ;

```

If the $\langle \text{digit } 1 \rangle$ is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if $\langle \text{digit } 1 \rangle$ is zero, we care about digits 17 and 18, and whether further digits are zero.

```

14682 \cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
14683 {
14684   \if_meaning:w 0 #3
14685     \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
14686   \else:
14687     \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
14688   \fi:
14689   #1 #3
14690 }

```

(End definition for __fp_mul_significand_test_f:NNN.)

__fp_mul_significand_large_f:NwwNNNN In this branch, $\langle \text{digit } 1 \rangle$ is non-zero. The result is thus $\langle \text{digits } 1-16 \rangle$, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here, __fp_round_digit:Nw takes digits 17 and further (as an integer expression), and replaces it by a $\langle \text{rounding digit} \rangle$, suitable for __fp_round:NNN.

```

14691 \cs_new:Npn \__fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
14692 {
14693   \exp_after:wN \__fp_basics_pack_high:NNNNNw
14694   \int_value:w \__fp_int_eval:w 1#2
14695   \exp_after:wN \__fp_basics_pack_low:NNNNNw
14696   \int_value:w \__fp_int_eval:w 1#3#4#5#6#7
14697   + \exp_after:wN \__fp_round:NNN
14698   \exp_after:wN #1
14699   \exp_after:wN #7
14700   \int_value:w \__fp_round_digit:Nw
14701 }

```

(End definition for __fp_mul_significand_large_f:NwwNNNN.)

__fp_mul_significand_small_f:NNwwwN In this branch, $\langle \text{digit } 1 \rangle$ is zero. Our result is thus $\langle \text{digits } 2-17 \rangle$, plus some rounding which depends on the digits 17, 18, and whether all subsequent digits are zero or not. The 8 digits 1#3 are followed, after expansion of the small_pack auxiliary, by the next digit, to form a 9 digit number.

```

14702 \cs_new:Npn \__fp_mul_significand_small_f:NNwwwN #1 #2#3; #4#5; #6; + #7
14703 {
14704   - 1
14705   \exp_after:wN \__fp_basics_pack_high:NNNNNw
14706   \int_value:w \__fp_int_eval:w 1#3#4

```

```

14707     \exp_after:wN \__fp_basics_pack_low:NNNNNw
14708     \int_value:w \__fp_int_eval:w 1#5#6#7
14709     + \exp_after:wN \__fp_round:NNN
14710     \exp_after:wN #1
14711     \exp_after:wN #7
14712     \int_value:w \__fp_round_digit:Nw
14713 }

```

(End definition for __fp_mul_significand_small_f:NNwwN.)

28.3 Division

28.3.1 Signs, and special numbers

Time is now ripe to tackle the hardest of the four elementary operations: division.

__fp/_o:ww Filtering special floating point is very similar to what we did for multiplications, with a few variations. Invalid operation exceptions display / rather than *. In the formula for dispatch, we replace - 2 + by -. The case of normal numbers is treated using __fp_div_npos_o:Nww rather than __fp_mul_npos_o:Nww. There are two additional cases: if the first operand is normal and the second is a zero, then the division by zero exception is raised: cases 10 and 11 of the \if_case:w construction in __fp_mul_cases_o:NnNww are provided as the fourth argument here.

```

14714 \cs_new:cpn { __fp/_o:ww }
14715 {
14716     \__fp_mul_cases_o:NnNww
14717     /
14718     { - }
14719     \__fp_div_npos_o:Nww
14720     {
14721         \or:
14722         \__fp_case_use:nw
14723         { \__fp_division_by_zero_o:NNww \c_inf_fp / }
14724         \or:
14725         \__fp_case_use:nw
14726         { \__fp_division_by_zero_o:NNww \c_minus_inf_fp / }
14727     }
14728 }

```

(End definition for __fp/_o:ww.)

```

\__fp_div_npos_o:Nww \__fp_div_npos_o:Nww <final sign> \s__fp \__fp_chk:w 1 <sign_A> {\exp A}
{\langle A_1 \rangle} {\langle A_2 \rangle} {\langle A_3 \rangle} {\langle A_4 \rangle} ; \s__fp \__fp_chk:w 1 <sign_Z> {\exp Z}
{\langle Z_1 \rangle} {\langle Z_2 \rangle} {\langle Z_3 \rangle} {\langle Z_4 \rangle} ;

```

We want to compute A/Z . As for multiplication, __fp_sanitize:Nw checks for overflow or underflow; we provide it with the $\langle final\ sign \rangle$, and an integer expression in which we compute the exponent. We set up the arguments of __fp_div_significand_i_o:wnnw, namely an integer $\langle y \rangle$ obtained by adding 1 to the first 5 digits of Z (explanation given soon below), then the four $\{\langle A_i \rangle\}$, then the four $\{\langle Z_i \rangle\}$, a semi-colon, and the $\langle final\ sign \rangle$, used for rounding at the end.

```

14729 \cs_new:Npn \__fp_div_npos_o:Nww
14730 #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
14731 {

```

```

14732     \exp_after:wN \__fp_sanitizew
14733     \exp_after:wN #1
14734     \int_value:w \__fp_int_eval:w
14735     #3 - #6
14736     \exp_after:wN \__fp_div_significand_i_o:wnnw
14737     \int_value:w \__fp_int_eval:w #7 \use_i:nnnn #8 + 1 ;
14738     #4
14739     {#7}{#8}#9 ;
14740     #1
14741 }

```

(End definition for `__fp_div_npos_o:Nww`.)

28.3.2 Work plan

In this subsection, we explain how to avoid overflowing \TeX 's integers when performing the division of two positive normal numbers.

We are given two numbers, $A = 0.A_1A_2A_3A_4$ and $Z = 0.Z_1Z_2Z_3Z_4$, in blocks of 4 digits, and we know that the first digits of A_1 and of Z_1 are non-zero. To compute A/Z , we proceed as follows.

- Find an integer $Q_A \simeq 10^4 A/Z$.
- Replace A by $B = 10^4 A - Q_A Z$.
- Find an integer $Q_B \simeq 10^4 B/Z$.
- Replace B by $C = 10^4 B - Q_B Z$.
- Find an integer $Q_C \simeq 10^4 C/Z$.
- Replace C by $D = 10^4 C - Q_C Z$.
- Find an integer $Q_D \simeq 10^4 D/Z$.
- Consider $E = 10^4 D - Q_D Z$, and ensure correct rounding.

The result is then $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$. Since the Q_i are integers, B , C , D , and E are all exact multiples of 10^{-16} , in other words, computing with 16 digits after the decimal separator yields exact results. The problem is the risk of overflow: in general B , C , D , and E may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that $Q_A \leq 10^4 A/Z$ etc. A reasonable attempt would be to define Q_A as

$$\text{\int_eval:n} \left\{ \frac{A_1 A_2}{Z_1 + 1} - 1 \right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that $\varepsilon\text{-}\text{\TeX}$'s `__fp_int_eval:w` rounds divisions instead of truncating (really, $1/2$ would be sufficient, but we work with integers). We add 1 to Z_1 because $Z_1 \leq 10^4 Z < Z_1 + 1$ and we need Q_A to be an underestimate. However, we are now underestimating Q_A too much: it can be wrong by up to 100, for instance when $Z = 0.1$ and $A \simeq 1$. Then B could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since \TeX can only handle integers less than roughly $2 \cdot 10^9$.

A better formula is to take

$$Q_A = \backslash\text{int_eval:n}\left\{\frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1\right\}.$$

This is always less than $10^9 A / (10^5 Z)$, as we wanted. In words, we take the 5 first digits of Z into account, and the 8 first digits of A , using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the Q_i avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that $\varepsilon\text{-TeX}$ rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31} - 1)/10^9 \simeq 2.147 \dots$, since the numerator involves an integer slightly smaller than $10^9 A$.

Let us bound B :

$$\begin{aligned} 10^5 B &= A_1 A_2 0 + 10 \cdot 0.A_3 A_4 - 10 \cdot Z_1.Z_2 Z_3 Z_4 \cdot Q_A \\ &< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1.Z_2 Z_3 Z_4}{y}\right) + \frac{3}{2} \cdot 10 \cdot Z_1.Z_2 Z_3 Z_4 + 10 \\ &\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1.Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10 \\ &\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y. \end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned} 10^5 B &< 10^9 A / y + 1.6y, \\ 10^5 C &< 10^9 B / y + 1.6y, \\ 10^5 D &< 10^9 C / y + 1.6y, \\ 10^5 E &< 10^9 D / y + 1.6y. \end{aligned}$$

The goal is now to prove that none of B , C , D , and E can go beyond $(2^{31} - 1)/10^9 = 2.147 \dots$.

Combining the various inequalities together with $A < 1$, we get

$$\begin{aligned} 10^5 B &< 10^9 / y + 1.6y, \\ 10^5 C &< 10^{13} / y^2 + 1.6(y + 10^4), \\ 10^5 D &< 10^{17} / y^3 + 1.6(y + 10^4 + 10^8 / y), \\ 10^5 E &< 10^{21} / y^4 + 1.6(y + 10^4 + 10^8 / y + 10^{12} / y^2). \end{aligned}$$

All of those bounds are convex functions of y (since every power of y involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \leq 10^5$. Thus,

$$\begin{aligned} 10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \\ 10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \\ 10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \\ 10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5). \end{aligned}$$

All of those bounds are less than $2.147 \cdot 10^5$, and we are thus within $\text{T}_{\text{E}}\text{X}$'s bounds in all cases!

We later need to have a bound on the Q_i . Their definitions imply that $Q_A < 10^9 A/y - 1/2 < 10^5 A$ and similarly for the other Q_i . Thus, all of them are less than 177770.

The last step is to ensure correct rounding. We have

$$A/Z = \sum_{i=1}^4 (10^{-4i} Q_i) + 10^{-16} E/Z$$

exactly. Furthermore, we know that the result is in $[0.1, 10)$, hence will be rounded to a multiple of 10^{-16} or of 10^{-15} , so we only need to know the integer part of E/Z , and a “rounding” digit encoding the rest. Equivalently, we need to find the integer part of $2E/Z$, and determine whether it was an exact integer or not (this serves to detect ties). Since

$$\frac{2E}{Z} = 2 \frac{10^5 E}{10^5 Z} \leq 2 \frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let $\varepsilon\text{-T}_{\text{E}}\text{X}$ round

$$P = \backslash\text{int_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

which differs from $2E/Z$ by at most

$$\frac{1}{2} + 2 \left| \frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2} \right| + 2 \left| \frac{10^8 E - E_1 E_2}{Z_1 Z_2} \right| < 1,$$

($1/2$ comes from $\varepsilon\text{-T}_{\text{E}}\text{X}$'s rounding) because each absolute value is less than 10^{-7} . Thus P is either the correct integer part, or is off by 1; furthermore, if $2E/Z$ is an integer, $P = 2E/Z$. We will check the sign of $2E - PZ$. If it is negative, then $E/Z \in ((P-1)/2, P/2)$. If it is zero, then $E/Z = P/2$. If it is positive, then $E/Z \in (P/2, (P+1)/2)$. In each case, we know how to round to an integer, depending on the parity of P , and the rounding mode.

28.3.3 Implementing the significand division

`_fp_div_significand_i_o:wnnw`

`_fp_div_significand_i_o:wnnw <y> ; {<A1>} {<A2>} {<A3>} {<A4>}
{<Z1>} {<Z2>} {<Z3>} {<Z4>} ; <sign>`

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace $\langle A_1 \rangle$ and $\langle A_2 \rangle$, and prepare the $\langle continuation \rangle$ arguments for 4 consecutive calls to `_fp_div_significand_calc:wnnnnnnn`. Each of these calls needs $\langle y \rangle$ ($\#1$), and it turns out that

we need post-expansion there, hence the `\int_value:w`. Here, `#4` is six brace groups, which give the six first n-type arguments of the `calc` function.

```

14742 \cs_new:Npn \__fp_div_significand_i_o:wnnw #1 ; #2#3 #4 ;
14743 {
14744   \exp_after:wN \__fp_div_significand_test_o:w
14745   \int_value:w \__fp_int_eval:w
14746   \exp_after:wN \__fp_div_significand_calc:wnnnnnnn
14747   \int_value:w \__fp_int_eval:w 999999 + #2 #3 0 / #1 ;
14748   #2 #3 ;
14749   #4
14750   { \exp_after:wN \__fp_div_significand_ii:wN \int_value:w #1 }
14751   { \exp_after:wN \__fp_div_significand_ii:wN \int_value:w #1 }
14752   { \exp_after:wN \__fp_div_significand_ii:wN \int_value:w #1 }
14753   { \exp_after:wN \__fp_div_significand_iii:wnnnnnn \int_value:w #1 }
14754 }

```

(End definition for `__fp_div_significand_i_o:wnnw`.)

```

\__fp_div_significand_calc:wnnnnnnn \__fp_div_significand_calc:wnnnnnnn <106 + QA> ; <A1> <A2> ; {<A3>}
\__fp_div_significand_calc_i:wnnnnnnn {<A4>} {<Z1>} {<Z2>} {<Z3>} {<Z4>} {<continuation>}
\__fp_div_significand_calc_ii:wnnnnnnn expands to
<106 + QA> <continuation> ; <B1> <B2> ; {<B3>} {<B4>} {<Z1>} {<Z2>} {<Z3>}
{<Z4>}

```

where $B = 10^4 A - Q_A \cdot Z$. This function is also used to compute C , D , E (with the input shifted accordingly), and is used in `l3fp-expo`.

We know that $0 < Q_A < 1.8 \cdot 10^5$, so the product of Q_A with each Z_i is within $\text{T}_\text{E}\text{X}$'s bounds. However, it is a little bit too large for our purposes: we would not be able to use the usual trick of adding a large power of 10 to ensure that the number of digits is fixed.

The bound on Q_A , implies that $10^6 + Q_A$ starts with the digit 1, followed by 0 or 1. We test, and call different auxiliaries for the two cases. An earlier implementation did the tests within the computation, but since we added a `<continuation>`, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$\begin{aligned}
& 10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7) \\
& + 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),
\end{aligned}$$

where $\langle i \rangle$ stands for the 10^5 digit of Q_A , which is 0 or 1, and $\#1$, $\#2$, *etc.* are the parameters of either auxiliary. The factors of 10 come from the fact that $Q_A = 10 \cdot 10^4 \cdot \langle i \rangle + \#1$. As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most 10^8 and the negative contributions can go up to 10^9 . Indeed, for the auxiliary with $\langle i \rangle = 1$, $\#1$ is at most 80000, leading to contributions of at worst $-8 \cdot 10^8 4$, while the other negative term is very small $< 10^6$ (except in the first expression, where we don't care about the number of digits); for the auxiliary with $\langle i \rangle = 0$, $\#1$ can go up to 99999, but there is no other negative term. Hence, a good choice is $2 \cdot 10^9$, which produces totals in the range $[10^9, 2.1 \cdot 10^9]$. We are flirting with $\text{T}_\text{E}\text{X}$'s limits once more.

```

14755 \cs_new:Npn \__fp_div_significand_calc:wnnnnnnn #1#

```



```

14756 {
14757   \if_meaning:w 1 #1
14758   \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnnn
14759   \else:
14760     \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnnn
14761   \fi:
14762 }
14763 \cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnnn
14764 #1; #2;#3#4 #5#6#7#8 #9
14765 {
14766   1 1 #1
14767   #9 \exp_after:wN ;
14768   \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
14769   + #2 - #1 * #5 - #5#60
14770   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14771   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
14772   + #3 - #1 * #6 - #70
14773   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14774   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
14775   + #4 - #1 * #7 - #80
14776   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14777   \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
14778   - #1 * #8 ;
14779   {#5}{#6}{#7}{#8}
14780 }
14781 \cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnnn
14782 #1; #2;#3#4 #5#6#7#8 #9
14783 {
14784   1 0 #1
14785   #9 \exp_after:wN ;
14786   \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
14787   + #2 - #1 * #5
14788   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14789   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
14790   + #3 - #1 * #6
14791   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14792   \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
14793   + #4 - #1 * #7
14794   \exp_after:wN \__fp_pack_Bigg:NNNNNNw
14795   \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
14796   - #1 * #8 ;
14797   {#5}{#6}{#7}{#8}
14798 }

```

(End definition for __fp_div_significand_calc:wwnnnnnnnn, __fp_div_significand_calc_i:wwnnnnnnnn,
and __fp_div_significand_calc_ii:wwnnnnnnnn.)

__fp_div_significand_ii:wwn __fp_div_significand_ii:wwn $\langle y \rangle$; $\langle B_1 \rangle$; $\{\langle B_2 \rangle\}$ $\{\langle B_3 \rangle\}$ $\{\langle B_4 \rangle\}$ $\{\langle Z_1 \rangle\}$
 $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$ $\langle continuations \rangle$ $\langle sign \rangle$

Compute Q_B by evaluating $\langle B_1 \rangle \langle B_2 \rangle 0 / y - 1$. The result is output to the left, in an
 $__fp_int_eval:w$ which we start now. Once that is evaluated (and the other Q_i also,
since later expansions are triggered by this one), a packing auxiliary takes care of placing
the digits of Q_B in an appropriate way for the final addition to obtain Q . This auxiliary
is also used to compute Q_C and Q_D with the inputs C and D instead of B .

```

14799 \cs_new:Npn \__fp_div_significand_ii:wwn #1; #2;#3
14800 {
14801   \exp_after:wN \__fp_div_significand_pack:NNN
14802   \int_value:w \__fp_int_eval:w
14803   \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
14804   \int_value:w \__fp_int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
14805 }

```

(End definition for __fp_div_significand_ii:wwn.)

```

\__fp_div_significand_iii:wwnnnnn <y> ; <E1> ; {<E2>} {<E3>} {<E4>}
{<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>

```

We compute $P \simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. Note the first 0, which multiplies Q_D by 10: we later add (roughly) $5 \cdot P$, which amounts to adding $P/2 \simeq E/Z$ to Q_D , the appropriate correction from a hypothetical Q_E .

```

14806 \cs_new:Npn \__fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
14807 {
14808   0
14809   \exp_after:wN \__fp_div_significand_iv:wwnnnnnnn
14810   \int_value:w \__fp_int_eval:w ( 2 * #2 #3 ) / #6 #7 ; % <- P
14811   #2 ; {#3} {#4} {#5}
14812   {#6} {#7}
14813 }

```

(End definition for __fp_div_significand_iii:wwnnnnn.)

```

\__fp_div_significand_iv:wwnnnnnnn <P> ; <E1> ; {<E2>} {<E3>} {<E4>}
\__fp_div_significand_v:NNw {<Z1>} {<Z2>} {<Z3>} {<Z4>} <sign>
\__fp_div_significand_vi:Nw

```

This adds to the current expression ($10^7 + 10 \cdot Q_D$) a contribution of $5 \cdot P + \text{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D , with an extra *rounding* digit. This *rounding* digit is 0 or 5 if T does not contribute, *i.e.*, if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation $\#1 \cdot \#6\#7$ below does not cause an overflow: naively, P can be up to 35, and $\#6\#7$ up to 10^8 , but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on P , which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $\#6\#7 \leq 10^8 \cdot Z$, hence $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot \#8\#9$ may overflow, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use $+$ as a separator (ending integer expressions explicitly). T is negative if the first character is $-$, it is positive if the first character is neither 0 nor $-$. It is also positive if the first character is 0 and second argument of $__fp_div_significand_vi:Nw$, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

```

14814 \cs_new:Npn \__fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9
14815 {
14816   + 5 * #1
14817   \exp_after:wN \__fp_div_significand_vi:Nw
14818   \int_value:w \__fp_int_eval:w -20 + 2*#2#3 - #1*#6#7 +
14819   \exp_after:wN \__fp_div_significand_v:NN
14820   \int_value:w \__fp_int_eval:w 199980 + 2*#4 - #1*#8 +
14821   \exp_after:wN \__fp_div_significand_v:NN
14822   \int_value:w \__fp_int_eval:w 200000 + 2*#5 - #1*#9 ;
14823 }
14824 \cs_new:Npn \__fp_div_significand_v:NN #1#2 { #1#2 \__fp_int_eval_end: + }
14825 \cs_new:Npn \__fp_div_significand_vi:Nw #1#2;
14826 {
14827   \if_meaning:w 0 #1
14828     \if_int_compare:w \__fp_int_eval:w #2 > 0 + 1 \fi:
14829   \else:
14830     \if_meaning:w - #1 - \else: + \fi: 1
14831   \fi:
14832   ;
14833 }

```

(End definition for __fp_div_significand_iv:wwnnnnnnn, __fp_div_significand_v:NNw, and __fp_div_significand_vi:Nw.)

__fp_div_significand_pack:NNN At this stage, we are in the following situation: \TeX is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

$$\begin{aligned} & _ _ \text{fp_div_significand_test_o:w } 10^6 + Q_A _ _ \text{fp_div_significand_} \\ & \text{pack:NNN } 10^6 + Q_B _ _ \text{fp_div_significand_pack:NNN } 10^6 + Q_C _ _ \text{fp_} \\ & \text{div_significand_pack:NNN } 10^7 + 10 \cdot Q_D + 5 \cdot P + \varepsilon ; \langle \text{sign} \rangle \end{aligned}$$

Here, $\varepsilon = \text{sign}(T)$ is 0 in case $2E = PZ$, 1 in case $2E > PZ$, which means that P was the correct value, but not with an exact quotient, and -1 if $2E < PZ$, *i.e.*, P was an overestimate. The packing function we define now does nothing special: it removes the 10^6 and carries two digits (for the 10^5 's and the 10^4 's).

```

14834 \cs_new:Npn \__fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 ; }

```

(End definition for __fp_div_significand_pack:NNN.)

__fp_div_significand_test_o:w __fp_div_significand_test_o:w 1 0 $\langle 5d \rangle$; $\langle 4d \rangle$; $\langle 4d \rangle$; $\langle 5d \rangle$; $\langle \text{sign} \rangle$

The reason we know that the first two digits are 1 and 0 is that the final result is known to be between 0.1 (inclusive) and 10, hence \widetilde{Q}_A (the tilde denoting the contribution from the other Q_i) is at most 99999, and $10^6 + \widetilde{Q}_A = 10 \dots$.

It is now time to round. This depends on how many digits the final result will have.

```

14835 \cs_new:Npn \__fp_div_significand_test_o:w 10 #1
14836 {
14837   \if_meaning:w 0 #1
14838     \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
14839   \else:
14840     \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
14841   \fi:
14842   #1
14843 }

```

(End definition for __fp_div_significand_test_o:w.)

```

\__fp_div_significand_small_o:wwwNNNNwN \__fp_div_significand_small_o:wwwNNNNwN 0 <4d> ; <4d> ; <4d> ; <5d>
; <final sign>

```

Standard use of the functions `__fp_basics_pack_low:NNNNw` and `__fp_basics_pack_high:NNNNw`. We finally get to use the *<final sign>* which has been sitting there for a while.

```

14844 \cs_new:Npn \__fp_div_significand_small_o:wwwNNNNwN
14845 0 #1; #2; #3; #4#5#6#7#8; #9
14846 {
14847   \exp_after:wN \__fp_basics_pack_high:NNNNw
14848   \int_value:w \__fp_int_eval:w 1 #1#2
14849   \exp_after:wN \__fp_basics_pack_low:NNNNw
14850   \int_value:w \__fp_int_eval:w 1 #3#4#5#6#7
14851   + \__fp_round:NNN #9 #7 #8
14852   \exp_after:wN ;
14853 }

```

(End definition for `__fp_div_significand_small_o:wwwNNNNwN`.)

```

\__fp_div_significand_large_o:wwwNNNNwN \__fp_div_significand_large_o:wwwNNNNwN <5d> ; <4d> ; <4d> ; <5d> ;
<sign>

```

We know that the final result cannot reach 10, hence `1#1#2`, together with contributions from the level below, cannot reach $2 \cdot 10^9$. For rounding, we build the *<rounding digit>* from the last two of our 18 digits.

```

14854 \cs_new:Npn \__fp_div_significand_large_o:wwwNNNNwN
14855 #1; #2; #3; #4#5#6#7#8; #9
14856 {
14857   + 1
14858   \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNw
14859   \int_value:w \__fp_int_eval:w 1 #1 #2
14860   \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
14861   \int_value:w \__fp_int_eval:w 1 #3 #4 #5 #6 +
14862   \exp_after:wN \__fp_round:NNN
14863   \exp_after:wN #9
14864   \exp_after:wN #6
14865   \int_value:w \__fp_round_digit:Nw #7 #8 ;
14866   \exp_after:wN ;
14867 }

```

(End definition for `__fp_div_significand_large_o:wwwNNNNwN`.)

28.4 Square root

`__fp_sqrt_o:w` Zeros are unchanged: $\sqrt{-0} = -0$ and $\sqrt{+0} = +0$. Negative numbers (other than -0) have no real square root. Positive infinity, and `nan`, are unchanged. Finally, for normal positive numbers, there is some work to do.

```

14868 \cs_new:Npn \__fp_sqrt_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14869 {
14870   \if_meaning:w 0 #2 \__fp_case_return_same_o:w \fi:
14871   \if_meaning:w 2 #3
14872     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { sqrt } }
14873   \fi:
14874   \if_meaning:w 1 #2 \else: \__fp_case_return_same_o:w \fi:
14875   \__fp_sqrt_npos_o:w

```

```

14876     \s__fp \__fp_chk:w #2 #3 #4;
14877 }

```

(End definition for __fp_sqrt_o:w.)

```

\__fp_sqrt_npos_o:w
\__fp_sqrt_npos_auxi_o:w wnnN
\__fp_sqrt_npos_auxii_o:w wnnnnnnnn

```

Prepare __fp_sanitize:Nw to receive the final sign 0 (the result is always positive) and the exponent, equal to half of the exponent #1 of the argument. If the exponent #1 is even, find a first approximation of the square root of the significand $10^8 a_1 + a_2 = 10^8 \#2\#3 + \#4\#5$ through Newton's method, starting at $x = 57234133 \simeq 10^{7.75}$. Otherwise, first shift the significand of the argument by one digit, getting $a'_1 \in [10^6, 10^7)$ instead of $[10^7, 10^8)$, then use Newton's method starting at $17782794 \simeq 10^{7.25}$.

```

14878 \cs_new:Npn \__fp_sqrt_npos_o:w \s__fp \__fp_chk:w 1 0 #1#2#3#4#5;
14879 {
14880   \exp_after:wN \__fp_sanitize:Nw
14881   \exp_after:wN 0
14882   \int_value:w \__fp_int_eval:w
14883   \if_int_odd:w #1 \exp_stop_f:
14884     \exp_after:wN \__fp_sqrt_npos_auxi_o:w wnnN
14885     \fi:
14886     #1 / 2
14887     \__fp_sqrt_Newton_o:w wnn 56234133; 0; {#2#3} {#4#5} 0
14888 }
14889 \cs_new:Npn \__fp_sqrt_npos_auxi_o:w wnnN #1 / 2 #2; 0; #3#4#5
14890 {
14891   ( #1 + 1 ) / 2
14892   \__fp_pack_eight:w wnnnnnnnn
14893   \__fp_sqrt_npos_auxii_o:w wnnnnnnnn
14894   ;
14895   0 #3 #4
14896 }
14897 \cs_new:Npn \__fp_sqrt_npos_auxii_o:w wnnnnnnnn #1; #2#3#4#5#6#7#8#9
14898 { \__fp_sqrt_Newton_o:w wnn 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }

```

(End definition for __fp_sqrt_npos_o:w, __fp_sqrt_npos_auxi_o:w wnnN, and __fp_sqrt_npos_auxii_o:w wnnnnnnnn.)

```

\__fp_sqrt_Newton_o:w wnn

```

Newton's method maps $x \mapsto [(x + [10^8 a_1/x])/2]$ in each iteration, where $[b/c]$ denotes ε -TeX's division. This division rounds the real number b/c to the closest integer, rounding ties away from zero, hence when c is even, $b/c - 1/2 + 1/c \leq [b/c] \leq b/c + 1/2$ and when c is odd, $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2 - 1/(2c)$. For all c , $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2$.

Let us prove that the method converges when implemented with ε -TeX integer division, for any $10^6 \leq a_1 < 10^8$ and starting value $10^6 \leq x < 10^8$. Using the inequalities above and the arithmetic-geometric inequality $(x + t)/2 \geq \sqrt{xt}$ for $t = 10^8 a_1/x$, we find

$$x' = \left\lceil \frac{x + [10^8 a_1/x]}{2} \right\rceil \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}.$$

After any step of iteration, we thus have $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$. The new difference $\delta' = x' - \sqrt{10^8 a_1}$ after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \leq \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \leq \frac{\delta}{2} \frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4}.$$

For $\delta > 3/2$, this last expression is $\leq \delta/2 + 3/4 < \delta$, hence δ decreases at each step: since all x are integers, δ must reach a value $-1/4 < \delta \leq 3/2$. In this range of values, we get $\delta' \leq \frac{3}{4} \frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$. We deduce that the difference $\delta = x - \sqrt{10^8 a_1}$ eventually reaches a value in the interval $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 11.25 \cdot 10^{-8}]$, whose width is $1 + 11 \cdot 10^{-8}$. The corresponding interval for x may contain two integers, hence x might oscillate between those two values.

However, the fact that $x \mapsto x - 1$ and $x - 1 \mapsto x$ puts stronger constraints, which are not compatible: the first implies

$$x + [10^8 a_1 / x] \leq 2x - 2$$

hence $10^8 a_1 / x \leq x - 3/2$, while the second implies

$$x - 1 + [10^8 a_1 / (x - 1)] \geq 2x - 1$$

hence $10^8 a_1 / (x - 1) \geq x - 1/2$. Combining the two inequalities yields $x^2 - 3x/2 \geq 10^8 a_1 \geq x - 3x/2 + 1/2$, which cannot hold. Therefore, the iteration always converges to a single integer x . To stop the iteration when two consecutive results are equal, the function `_fp_sqrt_Newton_o:wnn` receives the newly computed result as `#1`, the previous result as `#2`, and a_1 as `#3`. Note that ε -TeX combines the computation of a multiplication and a following division, thus avoiding overflow in `#3 * 100000000 / #1`. In any case, the result is within $[10^7, 10^8]$.

```

14899 \cs_new:Npn \_fp_sqrt_Newton_o:wnn #1; #2; #3
14900 {
14901   \if_int_compare:w #1 = #2 \exp_stop_f:
14902     \exp_after:wN \_fp_sqrt_auxi_o:NNNNwnnnN
14903     \int_value:w \_fp_int_eval:w 9999 9999 +
14904     \exp_after:wN \_fp_use_none_until_s:w
14905   \fi:
14906   \exp_after:wN \_fp_sqrt_Newton_o:wnn
14907   \int_value:w \_fp_int_eval:w (#1 + #3 * 1 0000 0000 / #1) / 2 ;
14908   #1; {#3}
14909 }

```

(End definition for `_fp_sqrt_Newton_o:wnn`.)

`_fp_sqrt_auxi_o:NNNNwnnnN` This function is followed by $10^8 + x - 1$, which has 9 digits starting with 1, then ; $\{a_1\} \{a_2\} \{a'\}$. Here, $x \simeq \sqrt{10^8 a_1}$ and we want to estimate the square root of $a = 10^{-8} a_1 + 10^{-16} a_2 + 10^{-17} a'$. We set up an initial underestimate

$$y = (x - 1)10^{-8} + 0.2499998875 \cdot 10^{-8} \lesssim \sqrt{a}.$$

From the inequalities shown earlier, we know that $y \leq \sqrt{10^{-8} a_1} \leq \sqrt{a}$ and that $\sqrt{10^{-8} a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$ hence (using $0.1 \leq y \leq \sqrt{a} \leq 1$)

$$a - y^2 \leq 10^{-8} a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8},$$

and $\sqrt{a} - y = (a - y^2) / (\sqrt{a} + y) \leq 16 \cdot 10^{-8}$. Next, `_fp_sqrt_auxii_o:NNnnnnnnN` is called several times to get closer and closer underestimates of \sqrt{a} . By construction, the underestimates y are always increasing, $a - y^2 < 3.2 \cdot 10^{-8}$ for all. Also, $y < 1$.

```

14910 \cs_new:Npn \_fp_sqrt_auxi_o:NNNNwnnnN 1 #1#2#3#4#5;
14911 {

```

```

14912 \_fp_sqrt_auxii_o:NnnnnnnnN
14913 \_fp_sqrt_auxiii_o:wnnnnnnnn
14914 {#1#2#3#4} {#5} {2499} {9988} {7500}
14915 }

```

(End definition for _fp_sqrt_auxi_o:NNNNwnnnN.)

_fp_sqrt_auxii_o:NnnnnnnnN

This receives a continuation function #1, then five blocks of 4 digits for y , then two 8-digit blocks and a single digit for a . A common estimate of $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y)$ is $(a - y^2)/(2y)$, which leads to alternating overestimates and underestimates. We tweak this, to only work with underestimates (no need then to worry about signs in the computation). Each step finds the largest integer $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then computes the integer (with ε -TeX's rounding division)

$$10^{4j}z = \left[\left(\lfloor 10^{4j}(a - y^2) \rfloor - 257 \right) \cdot (0.5 \cdot 10^8) \right] / \lfloor 10^8 y + 1 \rfloor.$$

The choice of j ensures that $10^{4j}z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8 / 10^7 = 10^9$, thus $10^9 + 10^{4j}z$ has exactly 10 digits, does not overflow TeX's integer range, and starts with 1. Incidentally, since all $a - y^2 \leq 3.2 \cdot 10^{-8}$, we know that $j \geq 3$.

Let us show that z is an underestimate of $\sqrt{a} - y$. On the one hand, $\sqrt{a} - y \leq 16 \cdot 10^{-8}$ because this holds for the initial y and values of y can only increase. On the other hand, the choice of j implies that $\sqrt{a} - y \leq 5(\sqrt{a} + y)(\sqrt{a} - y) = 5(a - y^2) < 10^{9-4j}$. For $j = 3$, the first bound is better, while for larger j , the second bound is better. For all $j \in [3, 6]$, we find $\sqrt{a} - y < 16 \cdot 10^{-2j}$. From this, we deduce that

$$10^{4j}(\sqrt{a} - y) = \frac{10^{4j}(a - y^2 - (\sqrt{a} - y)^2)}{2y} \geq \frac{\lfloor 10^{4j}(a - y^2) \rfloor - 257}{2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor} + \frac{1}{2}$$

where we have replaced the bound $10^{4j}(16 \cdot 10^{-2j}) = 256$ by 257 and extracted the corresponding term $1/(2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor) \geq 1/2$. Given that ε -TeX's integer division obeys $\lfloor b/c \rfloor \leq b/c + 1/2$, we deduce that $10^{4j}z \leq 10^{4j}(\sqrt{a} - y)$, hence $y + z \leq \sqrt{a}$ is an underestimate of \sqrt{a} , as claimed. One implementation detail: because the computation involves $-4*4 - 2*3*5 - 2*2*6$ which may be as low as $-5 \cdot 10^8$, we need to use the `pack_big` functions, and the big shifts.

```

14916 \cs_new:Npn \_fp_sqrt_auxii_o:NnnnnnnnN #1 #2#3#4#5#6 #7#8#9
14917 {
14918   \exp_after:wN #1
14919   \int_value:w \_fp_int_eval:w \c__fp_big_leading_shift_int
14920   + #7 - #2 * #2
14921   \exp_after:wN \_fp_pack_big:NNNNNNw
14922   \int_value:w \_fp_int_eval:w \c__fp_big_middle_shift_int
14923   - 2 * #2 * #3
14924   \exp_after:wN \_fp_pack_big:NNNNNNw
14925   \int_value:w \_fp_int_eval:w \c__fp_big_middle_shift_int
14926   + #8 - #3 * #3 - 2 * #2 * #4
14927   \exp_after:wN \_fp_pack_big:NNNNNNw
14928   \int_value:w \_fp_int_eval:w \c__fp_big_middle_shift_int
14929   - 2 * #3 * #4 - 2 * #2 * #5
14930   \exp_after:wN \_fp_pack_big:NNNNNNw
14931   \int_value:w \_fp_int_eval:w \c__fp_big_middle_shift_int
14932   + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
14933   \exp_after:wN \_fp_pack_big:NNNNNNw

```

```

14934      \int_value:w \_fp_int_eval:w \c\_fp\_big\_middle\_shift\_int
14935      - 2 * #4 * #5 - 2 * #3 * #6
14936      \exp_after:wN \_fp\_pack\_big:NNNNNNw
14937      \int_value:w \_fp_int_eval:w \c\_fp\_big\_middle\_shift\_int
14938      - #5 * #5 - 2 * #4 * #6
14939      \exp_after:wN \_fp\_pack\_big:NNNNNNw
14940      \int_value:w \_fp_int_eval:w
14941      \c\_fp\_big\_middle\_shift\_int
14942      - 2 * #5 * #6
14943      \exp_after:wN \_fp\_pack\_big:NNNNNNw
14944      \int_value:w \_fp_int_eval:w
14945      \c\_fp\_big\_trailing\_shift\_int
14946      - #6 * #6 ;
14947      % (
14948      - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
14949      {#2}{#3}{#4}{#5}{#6} {#7}{#8}#9
14950      }

```

(End definition for _fp_sqrt_auxii_o:NnnnnnnnnN.)

```

\_fp\_sqrt\_auxiii\_o:wnnnnnnnnn
\_fp\_sqrt\_auxiv\_o:NNNNNNw
\_fp\_sqrt\_auxv\_o:NNNNNNw
\_fp\_sqrt\_auxvi\_o:NNNNNNw
\_fp\_sqrt\_auxvii\_o:NNNNNNw

```

We receive here the difference $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$, as $\langle d_2 \rangle$; $\{\langle d_3 \rangle\} \dots \{\langle d_{10} \rangle\}$, where each block has 4 digits, except $\langle d_2 \rangle$. This function finds the largest $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then leaves an open parenthesis and the integer $\lfloor 10^{4j}(a - y^2) \rfloor$ in an integer expression. The closing parenthesis is provided by the caller _fp_sqrt_auxii_o:NnnnnnnnnN, which completes the expression

$$10^{4j}z = \left[(\lfloor 10^{4j}(a - y^2) \rfloor - 257) \cdot (0.5 \cdot 10^8) / \lfloor 10^8 y + 1 \rfloor \right]$$

for an estimate of $10^{4j}(\sqrt{a} - y)$. If $d_2 \geq 2$, $j = 3$ and the **auxiv** auxiliary receives $10^{12}z$. If $d_2 \leq 1$ but $10^4 d_2 + d_3 \geq 2$, $j = 4$ and the **auxv** auxiliary is called, and receives $10^{16}z$, and so on. In all those cases, the **auxviii** auxiliary is set up to add z to y , then go back to the **auxii** step with continuation **auxiii** (the function we are currently describing). The maximum value of j is 6, regardless of whether $10^{12}d_2 + 10^8 d_3 + 10^4 d_4 + d_5 \geq 1$. In this last case, we detect when $10^{24}z < 10^7$, which essentially means $\sqrt{a} - y \lesssim 10^{-17}$: once this threshold is reached, there is enough information to find the correctly rounded \sqrt{a} with only one more call to _fp_sqrt_auxii_o:NnnnnnnnnN. Note that the iteration cannot be stuck before reaching $j = 6$, because for $j < 6$, one has $2 \cdot 10^8 \leq 10^{4(j+1)}(a - y^2)$, hence

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{\lfloor 10^8 y + 1 \rfloor} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

14951 \cs_new:Npn \_fp\_sqrt\_auxiii\_o:wnnnnnnnnn
14952   #1; #2#3#4#5#6#7#8#9
14953   {
14954     \if_int_compare:w #1 > 1 \exp_stop_f:
14955     \exp_after:wN \_fp\_sqrt\_auxiv\_o:NNNNNNw
14956     \int_value:w \_fp_int_eval:w (#1#2 %)
14957   \else:
14958     \if_int_compare:w #1#2 > 1 \exp_stop_f:
14959     \exp_after:wN \_fp\_sqrt\_auxv\_o:NNNNNNw
14960     \int_value:w \_fp_int_eval:w (#1#2#3 %)
14961   \else:
14962     \if_int_compare:w #1#2#3 > 1 \exp_stop_f:
14963     \exp_after:wN \_fp\_sqrt\_auxvi\_o:NNNNNNw

```



```

14964         \int_value:w \_fp_int_eval:w (#1#2#3#4 %)
14965     \else:
14966         \exp_after:wN \_fp_sqrt_auxvii_o:NNNNNw
14967         \int_value:w \_fp_int_eval:w (#1#2#3#4#5 %)
14968     \fi:
14969     \fi:
14970 \fi:
14971 }
14972 \cs_new:Npn \_fp_sqrt_auxiv_o:NNNNNw #1#2#3#4#5#6;
14973 { \_fp_sqrt_auxviii_o:nnnnnnnn {#1#2#3#4#5#6} {00000000} }
14974 \cs_new:Npn \_fp_sqrt_auxv_o:NNNNNw #1#2#3#4#5#6;
14975 { \_fp_sqrt_auxviii_o:nnnnnnnn {000#1#2#3#4#5} {#60000} }
14976 \cs_new:Npn \_fp_sqrt_auxvi_o:NNNNNw #1#2#3#4#5#6;
14977 { \_fp_sqrt_auxviii_o:nnnnnnnn {0000000#1} {#2#3#4#5#6} }
14978 \cs_new:Npn \_fp_sqrt_auxvii_o:NNNNNw #1#2#3#4#5#6;
14979 {
14980     \if_int_compare:w #1#2 = 0 \exp_stop_f:
14981     \exp_after:wN \_fp_sqrt_auxx_o:Nnnnnnnnn
14982     \fi:
14983     \_fp_sqrt_auxviii_o:nnnnnnnn {00000000} {000#1#2#3#4#5}
14984 }

```

(End definition for `_fp_sqrt_auxiii_o:nnnnnnnn` and others.)

`_fp_sqrt_auxviii_o:nnnnnnnn` Simply add the two 8-digit blocks of z , aligned to the last four of the five 4-digit blocks of y , then call the `auxii` auxiliary to evaluate $y'^2 = (y + z)^2$.

```

14985 \cs_new:Npn \_fp_sqrt_auxviii_o:nnnnnnnn #1#2 #3#4#5#6#7
14986 {
14987     \exp_after:wN \_fp_sqrt_auxix_o:wnwnw
14988     \int_value:w \_fp_int_eval:w #3
14989     \exp_after:wN \_fp_basics_pack_low:NNNNNw
14990     \int_value:w \_fp_int_eval:w #1 + 1#4#5
14991     \exp_after:wN \_fp_basics_pack_low:NNNNNw
14992     \int_value:w \_fp_int_eval:w #2 + 1#6#7 ;
14993 }
14994 \cs_new:Npn \_fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
14995 {
14996     \_fp_sqrt_auxii_o:NnnnnnnnnN
14997     \_fp_sqrt_auxiii_o:nnnnnnnnnn {#1}{#2}{#3}{#4}{#5}
14998 }

```

(End definition for `_fp_sqrt_auxviii_o:nnnnnnnn` and `_fp_sqrt_auxix_o:wnwnw`.)

`_fp_sqrt_auxx_o:Nnnnnnnnn` At this stage, $j = 6$ and $10^{24}z < 10^7$, hence
`_fp_sqrt_auxxi_o:wnnnN`

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq (10^{24}(a - y^2) - 258) \cdot (0.5 \cdot 10^8) / (10^8y + 1),$$

then $10^{24}(a - y^2) - 258 < 2(10^7 + 1/2)(y + 10^{-8})$, and

$$10^{24}(a - y^2) < (10^7 + 1290.5)(1 + 10^{-8}/y)(2y) < (10^7 + 1290.5)(1 + 10^{-7})(y + \sqrt{a}),$$

which finally implies $0 \leq \sqrt{a} - y < 0.2 \cdot 10^{-16}$. In particular, y is an underestimate of \sqrt{a} and $y + 0.5 \cdot 10^{-16}$ is a (strict) overestimate. There is at exactly one multiple m of $0.5 \cdot 10^{-16}$ in the interval $[y, y + 0.5 \cdot 10^{-16})$. If $m^2 > a$, then the square root is inexact and

is obtained by rounding $m - \epsilon$ to a multiple of 10^{-16} (the precise shift $0 < \epsilon < 0.5 \cdot 10^{-16}$ is irrelevant for rounding). If $m^2 = a$ then the square root is exactly m , and there is no rounding. If $m^2 < a$ then we round $m + \epsilon$. For now, discard a few irrelevant arguments #1, #2, #3, and find the multiple of $0.5 \cdot 10^{-16}$ within $[y, y + 0.5 \cdot 10^{-16})$; rather, only the last 4 digits #8 of y are considered, and we do not perform any carry yet. The `auxxi` auxiliary sets up `auxii` with a continuation function `auxxii` instead of `auxiii` as before. To prevent `auxii` from giving a negative results $a - m^2$, we compute $a + 10^{-16} - m^2$ instead, always positive since $m < \sqrt{a} + 0.5 \cdot 10^{-16}$ and $a \leq 1 - 10^{-16}$.

```

14999 \cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
15000 {
15001   \exp_after:wN \__fp_sqrt_auxxi_o:wwnnN
15002   \int_value:w \__fp_int_eval:w
15003     (#8 + 2499) / 5000 * 5000 ;
15004   {#4} {#5} {#6} {#7} ;
15005 }
15006 \cs_new:Npn \__fp_sqrt_auxxi_o:wwnnN #1; #2; #3#4#5
15007 {
15008   \__fp_sqrt_auxii_o:NnnnnnnnN
15009   \__fp_sqrt_auxxii_o:nnnnnnnnw
15010   #2 {#1}
15011   {#3} { #4 + 1 } #5
15012 }

```

(End definition for `__fp_sqrt_auxx_o:Nnnnnnnn` and `__fp_sqrt_auxxi_o:wwnnN`.)

`__fp_sqrt_auxxii_o:nnnnnnnnw`
`__fp_sqrt_auxxiii_o:w`

The difference $0 \leq a + 10^{-16} - m^2 \leq 10^{-16} + (\sqrt{a} - m)(\sqrt{a} + m) \leq 2 \cdot 10^{-16}$ was just computed: its first 8 digits vanish, as do the next four, #1, and most of the following four, #2. The guess m is an overestimate if $a + 10^{-16} - m^2 < 10^{-16}$, that is, #1#2 vanishes. Otherwise it is an underestimate, unless $a + 10^{-16} - m^2 = 10^{-16}$ exactly. For an underestimate, call the `auxxiv` function with argument 9998. For an exact result call it with 9999, and for an overestimate call it with 10000.

```

15013 \cs_new:Npn \__fp_sqrt_auxxii_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
15014 {
15015   \if_int_compare:w #1#2 > 0 \exp_stop_f:
15016   \if_int_compare:w #1#2 = 1 \exp_stop_f:
15017   \if_int_compare:w #3#4 = 0 \exp_stop_f:
15018   \if_int_compare:w #5#6 = 0 \exp_stop_f:
15019   \if_int_compare:w #7#8 = 0 \exp_stop_f:
15020     \__fp_sqrt_auxxiii_o:w
15021   \fi:
15022   \fi:
15023   \fi:
15024   \fi:
15025   \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
15026   \int_value:w 9998
15027 \else:
15028   \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
15029   \int_value:w 10000
15030 \fi:
15031 ;
15032 }
15033 \cs_new:Npn \__fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
15034 {

```

```

15035     \fi: \fi: \fi: \fi: \fi:
15036     \__fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
15037 }

```

(End definition for __fp_sqrt_auxxii_o:nnnnnnnnw and __fp_sqrt_auxxiii_o:w.)

__fp_sqrt_auxxiv_o:wnnnnnnnN

This receives 9998, 9999 or 10000 as #1 when m is an underestimate, exact, or an overestimate, respectively. Then comes m as five blocks of 4 digits, but where the last block #6 may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless m is an overestimate (#1 is then 10000). Then comes a as three arguments. Rounding is done by __fp_round:NNN, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are “rounded to even”, and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by __fp_round_digit:Nw, which receives (after removal of the 10000’s digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```

15038 \cs_new:Npn \__fp_sqrt_auxxiv_o:wnnnnnnnN #1; #2#3#4#5#6 #7#8#9
15039 {
15040     \exp_after:wN \__fp_basics_pack_high:NNNNNw
15041     \int_value:w \__fp_int_eval:w 1 0000 0000 + #2#3
15042     \exp_after:wN \__fp_basics_pack_low:NNNNNw
15043     \int_value:w \__fp_int_eval:w 1 0000 0000
15044     + #4#5
15045     \if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:
15046     + \exp_after:wN \__fp_round:NNN
15047     \exp_after:wN 0
15048     \exp_after:wN 0
15049     \int_value:w
15050     \exp_after:wN \use_i:nn
15051     \exp_after:wN \__fp_round_digit:Nw
15052     \int_value:w \__fp_int_eval:w #6 + 19999 - #1 ;
15053     \exp_after:wN ;
15054 }

```

(End definition for __fp_sqrt_auxxiv_o:wnnnnnnnN.)

28.5 About the sign

__fp_sign_o:w
__fp_sign_aux_o:w

Find the sign of the floating point: nan, +0, -0, +1 or -1.

```

15055 \cs_new:Npn \__fp_sign_o:w ? \s__fp \__fp_chk:w #1#2; @
15056 {
15057     \if_case:w #1 \exp_stop_f:
15058         \__fp_case_return_same_o:w
15059     \or: \exp_after:wN \__fp_sign_aux_o:w
15060     \or: \exp_after:wN \__fp_sign_aux_o:w
15061     \else: \__fp_case_return_same_o:w
15062     \fi:
15063     \s__fp \__fp_chk:w #1 #2;
15064 }
15065 \cs_new:Npn \__fp_sign_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 ;
15066 { \exp_after:wN \__fp_set_sign_o:w \exp_after:wN #2 \c_one_fp @ }

```

(End definition for `_fp_sign_o:w` and `_fp_sign_aux_o:w`.)

`_fp_set_sign_o:w` This function is used for the unary minus and for `abs`. It leaves the sign of `nan` invariant, turns negative numbers (sign 2) to positive numbers (sign 0) and positive numbers (sign 0) to positive or negative numbers depending on #1. It also expands after itself in the input stream, just like `_fp_+_o:ww`.

```

15067 \cs_new:Npn \_fp_set_sign_o:w #1 \s\_fp \_fp_chk:w #2#3#4; @
15068 {
15069   \exp_after:wN \_fp_exp_after_o:w
15070   \exp_after:wN \s\_fp
15071   \exp_after:wN \_fp_chk:w
15072   \exp_after:wN #2
15073   \int_value:w
15074   \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
15075   #4;
15076 }

```

(End definition for `_fp_set_sign_o:w`.)

28.6 Operations on tuples

`_fp_tuple_set_sign_o:w` Two cases: `abs(<tuple>)` for which #1 is 0 (invalid for tuples) and `-<tuple>` for which #1 is 2. In that case, map over all items in the tuple an auxiliary that dispatches to the type-appropriate sign-flipping function.

`_fp_tuple_set_sign_aux_o:Nnw`
`_fp_tuple_set_sign_aux_o:w`

```

15077 \cs_new:Npn \_fp_tuple_set_sign_o:w #1
15078 {
15079   \if_meaning:w 2 #1
15080     \exp_after:wN \_fp_tuple_set_sign_aux_o:Nnw
15081     \fi:
15082     \_fp_invalid_operation_o:nw { abs }
15083   }
15084 \cs_new:Npn \_fp_tuple_set_sign_aux_o:Nnw #1#2#3 @
15085 { \_fp_tuple_map_o:nw \_fp_tuple_set_sign_o:w #3 }
15086 \cs_new:Npn \_fp_tuple_set_sign_aux_o:w #1#2 ;
15087 {
15088   \_fp_change_func_type:NNN #1 \_fp_set_sign_o:w
15089   \_fp_parse_apply_unary_error:NNw
15090   2 #1 #2 ; @
15091 }

```

(End definition for `_fp_tuple_set_sign_o:w`, `_fp_tuple_set_sign_aux_o:Nnw`, and `_fp_tuple_set_sign_aux_o:w`.)

`_fp_*_tuple_o:ww` For `<number>*<tuple>` and `<tuple>*<number>` and `<tuple>/<number>`, loop through the `<tuple>` some code that multiplies or divides by the appropriate `<number>`. Importantly we need to dispatch according to the type, and we make sure to apply the operator in the correct order.

`_fp_tuple_*_o:ww`
`_fp_tuple_/_o:ww`

```

15092 \cs_new:cpn { \_fp\_*_tuple\_o:ww } #1 ;
15093 { \_fp_tuple_map_o:nw { \_fp_binary_type_o:Nww * #1 ; } }
15094 \cs_new:cpn { \_fp\_tuple\_*_o:ww } #1 ; #2 ;
15095 { \_fp_tuple_map_o:nw { \_fp_binary_rev_type_o:Nww * #2 ; } #1 ; }
15096 \cs_new:cpn { \_fp\_tuple\_/_o:ww } #1 ; #2 ;
15097 { \_fp_tuple_map_o:nw { \_fp_binary_rev_type_o:Nww / #2 ; } #1 ; }

```

(End definition for `_fp_tuple_o:ww`, `_fp_tuple_o:ww`, and `_fp_tuple_/_o:ww`.)

`_fp_tuple_+_tuple_o:ww` Check the two tuples have the same number of items and map through these a helper
`_fp_tuple__tuple_o:ww` that dispatches appropriately depending on the types. This means $(1,2)+((1,1),2)$
gives $(\text{nan},4)$.

```

15098 \cs_set_protected:Npn \_fp_tmp:w #1
15099   {
15100     \cs_new:cpn { \_fp_tuple_#1_tuple_o:ww }
15101       \s_fp_tuple \_fp_tuple_chk:w ##1 ;
15102       \s_fp_tuple \_fp_tuple_chk:w ##2 ;
15103     {
15104       \int_compare:nNnTF
15105         { \_fp_array_count:n {##1} } = { \_fp_array_count:n {##2} }
15106         { \_fp_tuple_mapthread_o:nww { \_fp_binary_type_o:Nww #1 } }
15107         { \_fp_invalid_operation_o:nww #1 }
15108       \s_fp_tuple \_fp_tuple_chk:w {##1} ;
15109       \s_fp_tuple \_fp_tuple_chk:w {##2} ;
15110     }
15111   }
15112 \_fp_tmp:w +
15113 \_fp_tmp:w -

```

(End definition for `_fp_tuple_+_tuple_o:ww` and `_fp_tuple__tuple_o:ww`.)

15114 \langle /initex | package \rangle

29 13fp-extended implementation

15115 \langle *initex | package \rangle

15116 \langle @@=fp \rangle

29.1 Description of fixed point numbers

This module provides a few functions to manipulate positive floating point numbers with extended precision (24 digits), but mostly provides functions for fixed-point numbers with this precision (24 digits). Those are used in the computation of Taylor series for the logarithm, exponential, and trigonometric functions. Since we eventually only care about the 16 first digits of the final result, some of the calculations are not performed with the full 24-digit precision. In other words, the last two blocks of each fixed point number may be wrong as long as the error is small enough to be rounded away when converting back to a floating point number. The fixed point numbers are expressed as

$$\{\langle a_1 \rangle\} \{\langle a_2 \rangle\} \{\langle a_3 \rangle\} \{\langle a_4 \rangle\} \{\langle a_5 \rangle\} \{\langle a_6 \rangle\} ;$$

where each $\langle a_i \rangle$ is exactly 4 digits (ranging from 0000 to 9999), except $\langle a_1 \rangle$, which may be any “not-too-large” non-negative integer, with or without leading zeros. Here, “not-too-large” depends on the specific function (see the corresponding comments for details). Checking for overflow is the responsibility of the code calling those functions. The fixed point number a corresponding to the representation above is $a = \sum_{i=1}^6 \langle a_i \rangle \cdot 10^{-4i}$.

Most functions we define here have the form

`_fp_fixed_<calculation>:wnn <operand1> ; <operand2> ; {<continuation>}`

They perform the $\langle calculation \rangle$ on the two $\langle operands \rangle$, then feed the result (6 brace groups followed by a semicolon) to the $\langle continuation \rangle$, responsible for the next step of the calculation. Some functions only accept an N-type $\langle continuation \rangle$. This allows constructions such as

```

    \__fp_fixed_add:wnn  $\langle X_1 \rangle$  ;  $\langle X_2 \rangle$  ;
    \__fp_fixed_mul:wnn  $\langle X_3 \rangle$  ;
    \__fp_fixed_add:wnn  $\langle X_4 \rangle$  ;

```

to compute $(X_1 + X_2) \cdot X_3 + X_4$. This turns out to be very appropriate for computing continued fractions and Taylor series.

At the end of the calculation, the result is turned back to a floating point number using `__fp_fixed_to_float_o:wn`. This function has to change the exponent of the floating point number: it must be used after starting an integer expression for the overall exponent of the result.

29.2 Helpers for numbers with extended precision

`\c__fp_one_fixed_tl` The fixed-point number 1, used in l3fp-expo.

```

15117 \tl_const:Nn \c__fp_one_fixed_tl
15118 { {10000} {0000} {0000} {0000} {0000} {0000} ; }

```

(End definition for `\c__fp_one_fixed_tl`.)

`__fp_fixed_continue:wn` This function simply calls the next function.

```

15119 \cs_new:Npn \__fp_fixed_continue:wn #1; #2 { #2 #1; }

```

(End definition for `__fp_fixed_continue:wn`.)

`__fp_fixed_add_one:wn` `__fp_fixed_add_one:wn $\langle a \rangle$; $\langle continuation \rangle$`

This function adds 1 to the fixed point $\langle a \rangle$, by changing a_1 to $10000 + a_1$, then calls the $\langle continuation \rangle$. This requires $a_1 + 10000 < 2^{31}$.

```

15120 \cs_new:Npn \__fp_fixed_add_one:wn #1#2; #3
15121 {
15122   \exp_after:wn #3 \exp_after:wn
15123   { \int_value:w \__fp_int_eval:w \c__fp_myriad_int + #1 } #2 ;
15124 }

```

(End definition for `__fp_fixed_add_one:wn`.)

`__fp_fixed_div_myriad:wn` Divide a fixed point number by 10000. This is a little bit more subtle than just removing the last group and adding a leading group of zeros: the first group #1 may have any number of digits, and we must split #1 into the new first group and a second group of exactly 4 digits. The choice of shifts allows #1 to be in the range $[0, 5 \cdot 10^8 - 1]$.

```

15125 \cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6;
15126 {
15127   \exp_after:wn \__fp_fixed_mul_after:wnn
15128   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
15129   \exp_after:wn \__fp_pack:NNNNNw
15130   \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
15131   + #1 ; {#2}{#3}{#4}{#5};
15132 }

```

(End definition for _fp_fixed_div_myriad:wn.)

_fp_fixed_mul_after:wn The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the $\langle continuation \rangle$ #3 in front.

```
15133 \cs_new:Npn \_fp_fixed_mul_after:wn #1; #2; #3 { #3 {#1} #2; }
```

(End definition for _fp_fixed_mul_after:wn.)

29.3 Multiplying a fixed point number by a short one

_fp_fixed_mul_short:wn
 $\{ \langle a_1 \rangle \} \{ \langle a_2 \rangle \} \{ \langle a_3 \rangle \} \{ \langle a_4 \rangle \} \{ \langle a_5 \rangle \} \{ \langle a_6 \rangle \} ;$
 $\{ \langle b_0 \rangle \} \{ \langle b_1 \rangle \} \{ \langle b_2 \rangle \} ; \{ \langle continuation \rangle \}$

Computes the product $c = ab$ of $a = \sum_i \langle a_i \rangle 10^{-4i}$ and $b = \sum_i \langle b_i \rangle 10^{-4i}$, rounds it to the closest multiple of 10^{-24} , and leaves $\langle continuation \rangle \{ \langle c_1 \rangle \} \dots \{ \langle c_6 \rangle \}$; in the input stream, where each of the $\langle c_i \rangle$ are blocks of 4 digits, except $\langle c_1 \rangle$, which is any TeX integer. Note that indices for $\langle b \rangle$ start at 0: for instance a second operand of $\{0001\}\{0000\}\{0000\}$ leaves the first operand unchanged (rather than dividing it by 10^4 , as _fp_fixed_mul:wn would).

```
15134 \cs_new:Npn \_fp_fixed_mul_short:wn #1#2#3#4#5#6; #7#8#9;
15135 {
15136   \exp_after:wN \_fp_fixed_mul_after:wn
15137   \int_value:w \_fp_int_eval:w \c__fp_leading_shift_int
15138   + #1*#7
15139   \exp_after:wN \_fp_pack:NNNNNw
15140   \int_value:w \_fp_int_eval:w \c__fp_middle_shift_int
15141   + #1*#8 + #2*#7
15142   \exp_after:wN \_fp_pack:NNNNNw
15143   \int_value:w \_fp_int_eval:w \c__fp_middle_shift_int
15144   + #1*#9 + #2*#8 + #3*#7
15145   \exp_after:wN \_fp_pack:NNNNNw
15146   \int_value:w \_fp_int_eval:w \c__fp_middle_shift_int
15147   + #2*#9 + #3*#8 + #4*#7
15148   \exp_after:wN \_fp_pack:NNNNNw
15149   \int_value:w \_fp_int_eval:w \c__fp_middle_shift_int
15150   + #3*#9 + #4*#8 + #5*#7
15151   \exp_after:wN \_fp_pack:NNNNNw
15152   \int_value:w \_fp_int_eval:w \c__fp_trailing_shift_int
15153   + #4*#9 + #5*#8 + #6*#7
15154   + ( #5*#9 + #6*#8 + #6*#9 / \c__fp_myriad_int )
15155   / \c__fp_myriad_int ; ;
15156 }
```

(End definition for _fp_fixed_mul_short:wn.)

29.4 Dividing a fixed point number by a small integer

_fp_fixed_div_int:wnN _fp_fixed_div_int:wnN $\langle a \rangle ; \langle n \rangle ; \langle continuation \rangle$

Divides the fixed point number $\langle a \rangle$ by the (small) integer $0 < \langle n \rangle < 10^4$ and feeds the result to the $\langle continuation \rangle$. There is no bound on a_1 .

The arguments of the i auxiliary are 1: one of the a_i , 2: n , 3: the ii or the iii auxiliary. It computes a (somewhat tight) lower bound Q_i for the ratio a_i/n .

The `ii` auxiliary receives Q_i , n , and a_i as arguments. It adds Q_i to a surrounding integer expression, and starts a new one with the initial value 9999, which ensures that the result of this expression has 5 digits. The auxiliary also computes $a_i - n \cdot Q_i$, placing the result in front of the 4 digits of a_{i+1} . The resulting $a'_{i+1} = 10^4(a_i - n \cdot Q_i) + a_{i+1}$ serves as the first argument for a new call to the `i` auxiliary.

When the `iii` auxiliary is called, the situation looks like this:

```

\__fp_fixed_div_int_after:Nw <continuation>
-1 + Q_1
\__fp_fixed_div_int_pack:Nw 9999 + Q_2
\__fp_fixed_div_int_pack:Nw 9999 + Q_3
\__fp_fixed_div_int_pack:Nw 9999 + Q_4
\__fp_fixed_div_int_pack:Nw 9999 + Q_5
\__fp_fixed_div_int_pack:Nw 9999
\__fp_fixed_div_int_auxii:wnn Q_6 ; {\langle n \rangle} {\langle a_6 \rangle}

```

where expansion is happening from the last line up. The `iii` auxiliary adds $Q_6 + 2 \simeq a_6/n + 1$ to the last 9999, giving the integer closest to $10000 + a_6/n$.

Each `pack` auxiliary receives 5 digits followed by a semicolon. The first digit is added as a carry to the integer expression above, and the 4 other digits are braced. Each call to the `pack` auxiliary thus produces one brace group. The last brace group is produced by the `after` auxiliary, which places the $\langle continuation \rangle$ as appropriate.

```

15157 \cs_new:Npn \__fp_fixed_div_int:wN #1#2#3#4#5#6 ; #7 ; #8
15158 {
15159   \exp_after:wN \__fp_fixed_div_int_after:Nw
15160   \exp_after:wN #8
15161   \int_value:w \__fp_int_eval:w - 1
15162   \__fp_fixed_div_int:wN
15163   #1; {\#7} \__fp_fixed_div_int_auxi:wN
15164   #2; {\#7} \__fp_fixed_div_int_auxi:wN
15165   #3; {\#7} \__fp_fixed_div_int_auxi:wN
15166   #4; {\#7} \__fp_fixed_div_int_auxi:wN
15167   #5; {\#7} \__fp_fixed_div_int_auxi:wN
15168   #6; {\#7} \__fp_fixed_div_int_auxii:wN ;
15169 }
15170 \cs_new:Npn \__fp_fixed_div_int:wN #1; #2 #3
15171 {
15172   \exp_after:wN #3
15173   \int_value:w \__fp_int_eval:w #1 / #2 - 1 ;
15174   {\#2}
15175   {\#1}
15176 }
15177 \cs_new:Npn \__fp_fixed_div_int_auxi:wN #1; #2 #3
15178 {
15179   + #1
15180   \exp_after:wN \__fp_fixed_div_int_pack:Nw
15181   \int_value:w \__fp_int_eval:w 9999
15182   \exp_after:wN \__fp_fixed_div_int:wN
15183   \int_value:w \__fp_int_eval:w #3 - #1*#2 \__fp_int_eval_end:
15184 }
15185 \cs_new:Npn \__fp_fixed_div_int_auxii:wN #1; #2 #3 { + #1 + 2 ; }
15186 \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {\#2} }
15187 \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {\#2} }

```


(End definition for `_fp_fixed_div_int:wwN` and others.)

29.5 Adding and subtracting fixed points

`_fp_fixed_add:wwn`
`_fp_fixed_sub:wwn`
`_fp_fixed_add:Nnnnnwnn`
`_fp_fixed_add:nnNnnwnn`
`_fp_fixed_add_pack:NNNNNwn`
`_fp_fixed_add_after:NNNNNwn`

`_fp_fixed_add:wwn` $\langle a \rangle$; $\langle b \rangle$; $\{\langle continuation \rangle\}$

Computes $a + b$ (resp. $a - b$) and feeds the result to the $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 \leq 114748$, its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits: $(a \pm b)_1 < 100000$. The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign, a_1, \dots, a_4 , the rest of a , and b_1 and b_2 . The second auxiliary receives the rest of a , the sign multiplying b , the rest of b , and the $\langle continuation \rangle$ as arguments. After going down through the various level, we go back up, packing digits and bringing the $\langle continuation \rangle$ (#8, then #7) from the end of the argument list to its start.

```

15188 \cs_new:Npn \_fp_fixed_add:wwn { \_fp_fixed_add:Nnnnnwnn + }
15189 \cs_new:Npn \_fp_fixed_sub:wwn { \_fp_fixed_add:Nnnnnwnn - }
15190 \cs_new:Npn \_fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
15191 {
15192   \exp_after:wN \_fp_fixed_add_after:NNNNNwn
15193   \int_value:w \_fp_int_eval:w 9 9999 9998 + #2#3 #1 #7#8
15194   \exp_after:wN \_fp_fixed_add_pack:NNNNNwn
15195   \int_value:w \_fp_int_eval:w 1 9999 9998 + #4#5
15196   \_fp_fixed_add:nnNnnwn #6 #1
15197 }
15198 \cs_new:Npn \_fp_fixed_add:nnNnnwn #1#2 #3 #4#5 #6#7 ; #8
15199 {
15200   #3 #4#5
15201   \exp_after:wN \_fp_fixed_add_pack:NNNNNwn
15202   \int_value:w \_fp_int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
15203 }
15204 \cs_new:Npn \_fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
15205 { + #1 ; {#7} {#2#3#4#5} {#6} }
15206 \cs_new:Npn \_fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
15207 { #7 {#1#2#3#4#5} {#6} }
```

(End definition for `_fp_fixed_add:wwn` and others.)

29.6 Multiplying fixed points

`_fp_fixed_mul:wwn`
`_fp_fixed_mul:nnnnnnnw`

`_fp_fixed_mul:wwn` $\langle a \rangle$; $\langle b \rangle$; $\{\langle continuation \rangle\}$

Computes $a \times b$ and feeds the result to $\langle continuation \rangle$. This function requires $0 \leq a_1, b_1 < 10000$. Once more, we need to play around the limit of 9 arguments for \TeX macros. Note that we don't need to obtain an exact rounding, contrarily to the $*$ operator,

so things could be harder. We wish to perform carries in

$$\begin{aligned}
a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where the $O(10^{-24})$ stands for terms which are at most $5 \cdot 10^{-24}$; ignoring those leads to an error of at most 5 ulp. Note how the first 15 terms only depend on a_1, \dots, a_4 and b_1, \dots, b_4 , while the last 6 terms only depend on a_1, a_2, a_5, a_6 , and the corresponding parts of b . Hence, the first function grabs a_1, \dots, a_4 , the rest of a , and b_1, \dots, b_4 , and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives $a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6$ and finally the $\langle continuation \rangle$ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The $\langle continuation \rangle$ is finally placed in front of the 6 brace groups by `__fp_fixed_mul_after:wnn`.

```

15208 \cs_new:Npn \__fp_fixed_mul:wnn #1#2#3#4 #5; #6#7#8#9
15209 {
15210   \exp_after:wN \__fp_fixed_mul_after:wnn
15211   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
15212   \exp_after:wN \__fp_pack:NNNNNw
15213   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
15214   + #1*#6
15215   \exp_after:wN \__fp_pack:NNNNNw
15216   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
15217   + #1*#7 + #2*#6
15218   \exp_after:wN \__fp_pack:NNNNNw
15219   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
15220   + #1*#8 + #2*#7 + #3*#6
15221   \exp_after:wN \__fp_pack:NNNNNw
15222   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
15223   + #1*#9 + #2*#8 + #3*#7 + #4*#6
15224   \exp_after:wN \__fp_pack:NNNNNw
15225   \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
15226   + #2*#9 + #3*#8 + #4*#7
15227   + ( #3*#9 + #4*#8
15228     + \__fp_fixed_mul:nnnnnnnw #5 {#6}{#7} {#1}{#2}
15229   )
15230 \cs_new:Npn \__fp_fixed_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
15231 {
15232   #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c__fp_myriad_int
15233   + #1*#3 + #5*#7 ; ;
15234 }
```

(End definition for `__fp_fixed_mul:wnn` and `__fp_fixed_mul:nnnnnnnw`.)

29.7 Combining product and sum of fixed points

`_fp_fixed_mul_add:wwwn`
`_fp_fixed_mul_sub_back:wwwn`
`_fp_fixed_one_minus_mul:wwn`
`_fp_fixed_mul_one_minus_mul:wwn`

`_fp_fixed_mul_add:wwwn` $\langle a \rangle ; \langle b \rangle ; \langle c \rangle ; \{\langle continuation \rangle\}$
`_fp_fixed_mul_sub_back:wwwn` $\langle a \rangle ; \langle b \rangle ; \langle c \rangle ; \{\langle continuation \rangle\}$
`_fp_fixed_one_minus_mul:wwn` $\langle a \rangle ; \langle b \rangle ; \{\langle continuation \rangle\}$

Compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the $\langle continuation \rangle$. Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of the computation of Taylor expansions, we over-optimize them a bit, and in particular we do not factor out the common parts of the three functions.

For definiteness, consider the task of computing $a \times b + c$. We perform carries in

$$\begin{aligned}
 a \times b + c = & (a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\
 & + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
 & + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\
 & + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
 & + \left(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
 & \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
 & \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 + c_5 c_6 \right) \cdot 10^{-24} + O(10^{-24}),
 \end{aligned}$$

where $c_1 c_2$, $c_3 c_4$, $c_5 c_6$ denote the 8-digit number obtained by juxtaposing the two blocks of digits of c , and \cdot denotes multiplication. The task is obviously tough because we have 18 brace groups in front of us.

Each of the three function starts the first two levels (the first, corresponding to 10^{-4} , is empty), with $c_1 c_2$ in the first level, calls the `i` auxiliary with arguments described later, and adds a trailing $+ c_5 c_6 ; \{\langle continuation \rangle\} ;$. The $+ c_5 c_6$ piece, which is omitted for `_fp_fixed_one_minus_mul:wwn`, is taken in the integer expression for the 10^{-24} level.

```

15235 \cs_new:Npn \\_fp_fixed_mul_add:wwwn #1; #2; #3#4#5#6#7#8;
15236 {
15237   \exp_after:wN \\_fp_fixed_mul_after:wwn
15238   \int_value:w \\_fp_int_eval:w \c__fp_big_leading_shift_int
15239   \exp_after:wN \\_fp_pack_big:NNNNNNw
15240   \int_value:w \\_fp_int_eval:w \c__fp_big_middle_shift_int + #3 #4
15241   \\_fp_fixed_mul_add:Nwnnnwnnn +
15242     + #5 #6 ; #2 ; #1 ; #2 ; +
15243     + #7 #8 ; ;
15244 }
15245 \cs_new:Npn \\_fp_fixed_mul_sub_back:wwwn #1; #2; #3#4#5#6#7#8;
15246 {
15247   \exp_after:wN \\_fp_fixed_mul_after:wwn
15248   \int_value:w \\_fp_int_eval:w \c__fp_big_leading_shift_int
15249   \exp_after:wN \\_fp_pack_big:NNNNNNw
15250   \int_value:w \\_fp_int_eval:w \c__fp_big_middle_shift_int + #3 #4
15251   \\_fp_fixed_mul_add:Nwnnnwnnn -
15252     + #5 #6 ; #2 ; #1 ; #2 ; -
15253     + #7 #8 ; ;
15254 }
15255 \cs_new:Npn \\_fp_fixed_one_minus_mul:wwn #1; #2;
15256 {
15257   \exp_after:wN \\_fp_fixed_mul_after:wwn

```

```

15258 \int_value:w \_fp_int_eval:w \c\_fp_big_leading_shift_int
15259 \exp_after:wN \_fp_pack_big:NNNNNNw
15260 \int_value:w \_fp_int_eval:w \c\_fp_big_middle_shift_int +
15261 1 0000 0000
15262 \_fp_fixed_mul_add:Nwnnnwnnn -
15263 ; #2 ; #1 ; #2 ; -
15264 ; ;
15265 }

```

(End definition for `_fp_fixed_mul_add:www`, `_fp_fixed_mul_sub_back:www`, and `_fp_fixed_mul_one_minus_mul:wn`.)

```

\_fp_fixed_mul_add:Nwnnnwnnn \_fp_fixed_mul_add:Nwnnnwnnn <op> + <c3> <c4> ;
<b> ; <a> ; <b> ; <op>
+ <c5> <c6> ;

```

Here, $\langle op \rangle$ is either + or -. Arguments #3, #4, #5 are $\langle b_1 \rangle$, $\langle b_2 \rangle$, $\langle b_3 \rangle$; arguments #7, #8, #9 are $\langle a_1 \rangle$, $\langle a_2 \rangle$, $\langle a_3 \rangle$. We can build three levels: $a_1 \cdot b_1$ for 10^{-8} , $(a_1 \cdot b_2 + a_2 \cdot b_1)$ for 10^{-12} , and $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$ for 10^{-16} . The $a-b$ products use the sign #1. Note that #2 is empty for `_fp_fixed_one_minus_mul:wn`. We call the *ii* auxiliary for levels 10^{-20} and 10^{-24} , keeping the pieces of $\langle a \rangle$ we've read, but not $\langle b \rangle$, since there is another copy later in the input stream.

```

15266 \cs_new:Npn \_fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
15267 {
15268   #1 #7*#3
15269   \exp_after:wN \_fp_pack_big:NNNNNNw
15270   \int_value:w \_fp_int_eval:w \c\_fp_big_middle_shift_int
15271   #1 #7*#4 #1 #8*#3
15272   \exp_after:wN \_fp_pack_big:NNNNNNw
15273   \int_value:w \_fp_int_eval:w \c\_fp_big_middle_shift_int
15274   #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
15275   \exp_after:wN \_fp_pack_big:NNNNNNw
15276   \int_value:w \_fp_int_eval:w \c\_fp_big_middle_shift_int
15277   #1 \_fp_fixed_mul_add:nnnnwnnnn {#7}{#8}{#9}
15278 }

```

(End definition for `_fp_fixed_mul_add:Nwnnnwnnn`.)

```

\_fp_fixed_mul_add:nnnnwnnnn \_fp_fixed_mul_add:nnnnwnnnn <a> ; <b> ; <op>
+ <c5> <c6> ;

```

Level 10^{-20} is $(a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1)$, multiplied by the sign, which was inserted by the *i* auxiliary. Then we prepare level 10^{-24} . We don't have access to all parts of $\langle a \rangle$ and $\langle b \rangle$ needed to make all products. Instead, we prepare the partial expressions

$$b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1$$

$$b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2.$$

Obviously, those expressions make no mathematical sense: we complete them with $a_5 \cdot$ and $\cdot b_5$, and with $a_6 \cdot b_1 + a_5 \cdot$ and $\cdot b_5 + a_1 \cdot b_6$, and of course with the trailing $+ c_5 c_6$. To do all this, we keep a_1 , a_5 , a_6 , and the corresponding pieces of $\langle b \rangle$.

```

15279 \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
15280 {
15281   ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )

```

```

15282 \exp_after:wN \_fp_pack_big:NNNNNNw
15283 \int_value:w \_fp_int_eval:w \c\_fp_big_trailing_shift_int
15284 \_fp_fixed_mul_add:nnnnwnnwN
15285 { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
15286 { #7 + #4*#8 + #3*#9 + #2 }
15287 {#1} #5;
15288 {#6}
15289 }

```

(End definition for _fp_fixed_mul_add:nnnnwnnwN.)

```

\_fp_fixed_mul_add:nnnnwnnwN {<partial1>} {<partial2>}
{<a1>} {<a5>} {<a6>} ; {<b1>} {<b5>} {<b6>} ;
<op> + <c5> <c6> ;

```

Complete the $\langle partial_1 \rangle$ and $\langle partial_2 \rangle$ expressions as explained for the `ii` auxiliary. The second one is divided by 10000: this is the carry from level 10^{-28} . The trailing $+ c_5 c_6$ is taken into the expression for level 10^{-24} . Note that the total of level 10^{-24} is in the interval $[-5 \cdot 10^8, 6 \cdot 10^8]$ (give or take a couple of 10000), hence adding it to the shift gives a 10-digit number, as expected by the packing auxiliaries. See `l3fp-aux` for the definition of the shifts and packing auxiliaries.

```

15290 \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
15291 {
15292   #9 (#4* #1 *#7)
15293   #9 (#5*#6+#4* #2 *#7+#3*#8) / \c\_fp_myriad_int
15294 }

```

(End definition for _fp_fixed_mul_add:nnnnwnnwN.)

29.8 Extended-precision floating point numbers

In this section we manipulate floating point numbers with roughly 24 significant figures (“extended-precision” numbers, in short, “ep”), which take the form of an integer exponent, followed by a comma, then six groups of digits, ending with a semicolon. The first group of digit may be any non-negative integer, while other groups of digits have 4 digits. In other words, an extended-precision number is an exponent ending in a comma, then a fixed point number. The corresponding value is $0.\langle digits \rangle \cdot 10^{\langle exponent \rangle}$. This convention differs from floating points.

_fp_ep_to_fixed:wnw Converts an extended-precision number with an exponent at most 4 and a first block less than 10^8 to a fixed point number whose first block has 12 digits, hopefully starting with many zeros.

```

15295 \cs_new:Npn \_fp_ep_to_fixed:wnw #1,#2
15296 {
15297   \exp_after:wN \_fp_ep_to_fixed_auxi:www
15298   \int_value:w \_fp_int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
15299   \exp:w \exp_end_continue_f:w
15300   \prg_replicate:nn { 4 - \int_max:nn {#1} { -32 } } { 0 } ;
15301 }
15302 \cs_new:Npn \_fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
15303 {
15304   \_fp_pack_eight:wNNNNNNNNN
15305   \_fp_pack_twice_four:wNNNNNNNNN
15306   \_fp_pack_twice_four:wNNNNNNNNN

```

```

15307     \__fp_pack_twice_four:wNNNNNNNN
15308     \__fp_ep_to_fixed_auxii:nnnnnnnwn ;
15309     #2 #1#3#4#5#6#7 0000 !
15310 }
15311 \cs_new:Npn \__fp_ep_to_fixed_auxii:nnnnnnnwn #1#2#3#4#5#6#7; #8! #9
15312 { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }

(End definition for \__fp_ep_to_fixed:wwn, \__fp_ep_to_fixed_auxi:www, and \__fp_ep_to_fixed_
auxii:nnnnnnnwn.)

```

```

\__fp_ep_to_ep:wwN
\__fp_ep_to_ep_loop:N
\__fp_ep_to_ep_end:www
\__fp_ep_to_ep_zero:ww

```

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation #8 is placed before the resulting exponent-mantissa pair. The input exponent may in fact be given as an integer expression. The `loop` auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the `end` auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with `__fp_use_i:ww` any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```

15313 \cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
15314 {
15315     \exp_after:wN #8
15316     \int_value:w \__fp_int_eval:w #1 + 4
15317     \exp_after:wN \use_i:nn
15318     \exp_after:wN \__fp_ep_to_ep_loop:N
15319     \int_value:w \__fp_int_eval:w 1 0000 0000 + #2 \__fp_int_eval_end:
15320     #3#4#5#6#7 ; ; !
15321 }
15322 \cs_new:Npn \__fp_ep_to_ep_loop:N #1
15323 {
15324     \if_meaning:w 0 #1
15325     - 1
15326     \else:
15327         \__fp_ep_to_ep_end:www #1
15328     \fi:
15329     \__fp_ep_to_ep_loop:N
15330 }
15331 \cs_new:Npn \__fp_ep_to_ep_end:www
15332 #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
15333 {
15334     \fi:
15335     \if_meaning:w ; #1
15336     - 2 * \c__fp_max_exponent_int
15337     \__fp_ep_to_ep_zero:ww
15338     \fi:
15339     \__fp_pack_twice_four:wNNNNNNNN
15340     \__fp_pack_twice_four:wNNNNNNNN
15341     \__fp_pack_twice_four:wNNNNNNNN
15342     \__fp_use_i:ww , ;
15343     #1 #2 0000 0000 0000 0000 0000 0000 ;
15344 }
15345 \cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;

```

```
15346 { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }
```

(End definition for `_fp_ep_to_ep:wwN` and others.)

```
\_fp_ep_compare:www
\_fp_ep_compare_aux:www
```

In l3fp-trig we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only 16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in [1000, 9999].

```
15347 \cs_new:Npn \_fp_ep_compare:www #1,#2#3#4#5#6#7;
15348 { \_fp_ep_compare_aux:www {#1}{#2}{#3}{#4}{#5}; #6#7; }
15349 \cs_new:Npn \_fp_ep_compare_aux:www #1;#2;#3,#4#5#6#7#8#9;
15350 {
15351   \if_case:w
15352     \_fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
15353     \if_int_compare:w #2 = #8#9 \exp_stop_f:
15354       0
15355     \else:
15356       \if_int_compare:w #2 < #8#9 - \fi: 1
15357     \fi:
15358   \or: 1
15359   \else: -1
15360   \fi:
15361 }
```

(End definition for `_fp_ep_compare:www` and `_fp_ep_compare_aux:www`.)

```
\_fp_ep_mul:wwwN
\_fp_ep_mul_raw:wwwN
```

Multiply two extended-precision numbers: first normalize them to avoid losing too much precision, then multiply the mantissas #2 and #4 as fixed point numbers, and sum the exponents #1 and #3. The result's first block is in [100, 9999].

```
15362 \cs_new:Npn \_fp_ep_mul:wwwN #1,#2; #3,#4;
15363 {
15364   \_fp_ep_to_ep:wwN #3,#4;
15365   \_fp_fixed_continue:wn
15366   {
15367     \_fp_ep_to_ep:wwN #1,#2;
15368     \_fp_ep_mul_raw:wwwN
15369   }
15370   \_fp_fixed_continue:wn
15371 }
15372 \cs_new:Npn \_fp_ep_mul_raw:wwwN #1,#2; #3,#4; #5
15373 {
15374   \_fp_fixed_mul:wn #2; #4;
15375   { \exp_after:wN #5 \int_value:w \_fp_int_eval:w #1 + #3 , }
15376 }
```

(End definition for `_fp_ep_mul:wwwN` and `_fp_ep_mul_raw:wwwN`.)

29.9 Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in l3fp-basics for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call $\langle n \rangle$ the numerator and $\langle d \rangle$ the denominator. After a simple normalization step, we can assume that $\langle n \rangle \in [0.1, 1)$ and $\langle d \rangle \in [0.1, 1)$, and compute $\langle n \rangle / (10 \langle d \rangle) \in$

(0.01, 1). In terms of the 6 blocks of digits $\langle n_1 \rangle \cdots \langle n_6 \rangle$ and the 6 blocks $\langle d_1 \rangle \cdots \langle d_6 \rangle$, the condition translates to $\langle n_1 \rangle, \langle d_1 \rangle \in [1000, 9999]$.

We first find an integer estimate $a \simeq 10^8 / \langle d \rangle$ by computing

$$\begin{aligned}\alpha &= \left\lfloor \frac{10^9}{\langle d_1 \rangle + 1} \right\rfloor \\ \beta &= \left\lfloor \frac{10^9}{\langle d_1 \rangle} \right\rfloor \\ a &= 10^3 \alpha + (\beta - \alpha) \cdot \left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) - 1250,\end{aligned}$$

where $\left\lfloor \frac{\cdot}{\cdot} \right\rfloor$ denotes ε -TEX's rounding division, which rounds ties away from zero. The idea is to interpolate between $10^3 \alpha$ and $10^3 \beta$ with a parameter $\langle d_2 \rangle / 10^4$, so that when $\langle d_2 \rangle = 0$ one gets $a = 10^3 \beta - 1250 \simeq 10^{12} / \langle d_1 \rangle \simeq 10^8 / \langle d \rangle$, while when $\langle d_2 \rangle = 9999$ one gets $a = 10^3 \alpha - 1250 \simeq 10^{12} / (\langle d_1 \rangle + 1) \simeq 10^8 / \langle d \rangle$. The shift by 1250 helps to ensure that a is an underestimate of the correct value. We shall prove that

$$1 - 1.755 \cdot 10^{-5} < \frac{\langle d \rangle a}{10^8} < 1.$$

We can then compute the inverse of $\langle d \rangle a / 10^8 = 1 - \epsilon$ using the relation $1 / (1 - \epsilon) \simeq (1 + \epsilon)(1 + \epsilon^2) + \epsilon^4$, which is correct up to a relative error of $\epsilon^5 < 1.6 \cdot 10^{-24}$. This allows us to find the desired ratio as

$$\frac{\langle n \rangle}{\langle d \rangle} = \frac{\langle n \rangle a}{10^8} ((1 + \epsilon)(1 + \epsilon^2) + \epsilon^4).$$

Let us prove the upper bound first (multiplied by 10^{15}). Note that $10^7 \langle d \rangle < 10^3 \langle d_1 \rangle + 10^{-1}(\langle d_2 \rangle + 1)$, and that ε -TEX's division $\left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor$ underestimates $10^{-1}(\langle d_2 \rangle + 1)$ by 0.5 at most, as can be checked for each possible last digit of $\langle d_2 \rangle$. Then,

$$10^7 \langle d \rangle a < \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \beta + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \alpha - 1250 \right) \quad (1)$$

$$< \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \quad (2)$$

$$\left(\left(10^3 - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \right) \left(\frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \quad (3)$$

$$< \left(10^3 \langle d_1 \rangle + \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor + \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lfloor \frac{\langle d_2 \rangle}{10} \right\rfloor \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 750 \right) \quad (4)$$

We recognize a quadratic polynomial in $[\langle d_2 \rangle / 10]$ with a negative leading coefficient: this polynomial is bounded above, according to $([\langle d_2 \rangle / 10] + a)(b - c[\langle d_2 \rangle / 10]) \leq (b + ca)^2 / (4c)$. Hence,

$$10^7 \langle d \rangle a < \frac{10^{15}}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} \left(\langle d_1 \rangle + \frac{1}{2} + \frac{1}{4} 10^{-3} - \frac{3}{8} \cdot 10^{-9} \langle d_1 \rangle (\langle d_1 \rangle + 1) \right)^2$$

Since $\langle d_1 \rangle$ takes integer values within $[1000, 9999]$, it is a simple programming exercise to check that the squared expression is always less than $\langle d_1 \rangle (\langle d_1 \rangle + 1)$, hence $10^7 \langle d \rangle a < 10^{15}$. The upper bound is proven. We also find that $\frac{3}{8}$ can be replaced by slightly smaller

numbers, but nothing less than 0.374563..., and going back through the derivation of the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7 \langle d \rangle a > \left(10^3 \langle d_1 \rangle + \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil - \frac{1}{2} \right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left\lceil \frac{\langle d_2 \rangle}{10} \right\rceil \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750 \right)$$

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values $[y/10] = 0$ or $[y/10] = 100$, and we easily check the bound for those values.

We have proven that the algorithm gives us a precise enough answer. Incidentally, the upper bound that we derived tells us that $a < 10^8 / \langle d \rangle \leq 10^9$, hence we can compute a safely as a $\text{T}_{\text{E}}\text{X}$ integer, and even add 10^9 to it to ease grabbing of all the digits. The lower bound implies $10^8 - 1755 < a$, which we do not care about.

`__fp_ep_div:wwwn` Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range $[100, 9999]$, and is placed after the $\langle continuation \rangle$ once we are done. First normalize the inputs so that both first block lie in $[1000, 9999]$, then call `__fp_ep_div_esti:wwwn` $\langle denominator \rangle$ $\langle numerator \rangle$, responsible for estimating the inverse of the denominator.

```

15377 \cs_new:Npn \__fp_ep_div:wwwn #1,#2; #3,#4;
15378 {
15379   \__fp_ep_to_ep:wwN #1,#2;
15380   \__fp_fixed_continue:wn
15381   {
15382     \__fp_ep_to_ep:wwN #3,#4;
15383     \__fp_ep_div_esti:wwwn
15384   }
15385 }
```

(End definition for `__fp_ep_div:wwwn`.)

`__fp_ep_div_esti:wwwn` The `esti` function evaluates $\alpha = 10^9 / (\langle d_1 \rangle + 1)$, which is used twice in the expression for a , and combines the exponents `#1` and `#4` (with a shift by 1 because we later compute $\langle n \rangle / (10 \langle d \rangle)$). Then the `estii` function evaluates $10^9 + a$, and puts the exponent `#2` after the continuation `#7`: from there on we can forget exponents and focus on the mantissa. The `estiii` function multiplies the denominator `#7` by $10^{-8}a$ (obtained as a split into the single digit `#1` and two blocks of 4 digits, `#2#3#4#5` and `#6`). The result $10^{-8}a \langle d \rangle = (1 - \epsilon)$, and a partially packed $10^{-9}a$ (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to `__fp_ep_div_epsilon:wnNNNNn`, which computes $10^{-9}a / (1 - \epsilon)$, that is, $1 / (10 \langle d \rangle)$ and we finally multiply this by the numerator `#8`.

```

15386 \cs_new:Npn \__fp_ep_div_esti:wwwn #1,#2#3; #4,
15387 {
15388   \exp_after:wN \__fp_ep_div_estii:wwnnwwn
15389   \int_value:w \__fp_int_eval:w 10 0000 0000 / ( #2 + 1 )
15390   \exp_after:wN ;
15391   \int_value:w \__fp_int_eval:w #4 - #1 + 1 ,
15392   {#2} #3;
15393 }
15394 \cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
```

```

15395 {
15396   \exp_after:wN \__fp_ep_div_estiii:NNNNNwwwn
15397   \int_value:w \__fp_int_eval:w 10 0000 0000 - 1750
15398   + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
15399   {#3}{#4}#5; #6; { #7 #2, }
15400 }
15401 \cs_new:Npn \__fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6; #7;
15402 {
15403   \__fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
15404   \__fp_ep_div_epsilon:wnNNNNNn {#1#2#3#4}#5#6
15405   \__fp_fixed_mul:wwn
15406 }

```

(End definition for __fp_ep_div_esti:wwwwn, __fp_ep_div_estii:wwnnwwn, and __fp_ep_div_estiii:NNNNNwwwn.)

```

\__fp_ep_div_epsilon:wnNNNNNn
\__fp_ep_div_eps_pack:NNNNNw
\__fp_ep_div_epsilon:wnNNNNNn

```

The bounds shown above imply that the `epsi` function's first operand is $(1 - \epsilon)$ with $\epsilon \in [0, 1.755 \cdot 10^{-5}]$. The `epsi` function computes ϵ as $1 - (1 - \epsilon)$. Since $\epsilon < 10^{-4}$, its first block vanishes and there is no need to explicitly use `#1` (which is 9999). Then `epsi` evaluates $10^{-9}a/(1 - \epsilon)$ as $(1 + \epsilon^2)(1 + \epsilon)(10^{-9}a\epsilon) + 10^{-9}a$. Importantly, we compute $10^{-9}a\epsilon$ before multiplying it with the rest, rather than multiplying by ϵ and then $10^{-9}a$, as this second option loses more precision. Also, the combination of `short_mul` and `div_myriad` is both faster and more precise than a simple `mul`.

```

15407 \cs_new:Npn \__fp_ep_div_epsilon:wnNNNNNn #1#2#3#4#5#6;
15408 {
15409   \exp_after:wN \__fp_ep_div_epsilon:wnNNNNNn
15410   \int_value:w \__fp_int_eval:w 1 9998 - #2
15411   \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
15412   \int_value:w \__fp_int_eval:w 1 9999 9998 - #3#4
15413   \exp_after:wN \__fp_ep_div_epsilon:wnNNNNNn
15414   \int_value:w \__fp_int_eval:w 2 0000 0000 - #5#6 ; ;
15415 }
15416 \cs_new:Npn \__fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
15417 { + #1 ; {#2#3#4#5} {#6} }
15418 \cs_new:Npn \__fp_ep_div_epsilon:wnNNNNNn 1#1; #2; #3#4#5#6#7#8
15419 {
15420   \__fp_fixed_mul:wwn {0000}{#1}#2; {0000}{#1}#2;
15421   \__fp_fixed_add_one:wN
15422   \__fp_fixed_mul:wwn {10000} {#1} #2 ;
15423   {
15424     \__fp_fixed_mul_short:wwn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
15425     \__fp_fixed_div_myriad:wn
15426     \__fp_fixed_mul:wwn
15427   }
15428   \__fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
15429 }

```

(End definition for __fp_ep_div_epsilon:wnNNNNNn, __fp_ep_div_eps_pack:NNNNNw, and __fp_ep_div_epsilon:wnNNNNNn.)

29.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 100 until we have $x \in [0.01, 1)$. Then find an integer approximation $r \in [101, 1003]$ of $10^2/\sqrt{x}$,

as the fixed point of iterations of the Newton method: essentially $r \mapsto (r + 10^8/(x_1 r))/2$, starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we replace 10^8 by a slightly larger number which ensures that $r^2 x \geq 10^4$. This also causes $r \in [101, 1003]$. Another correction to the above is that the input is actually normalized to $[0.1, 1)$, and we use either 10^8 or 10^9 in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation r , we set $y = 10^{-4} r^2 x$ (or rather, the correct power of 10 to get $y \simeq 1$) and compute $y^{-1/2}$ through another application of Newton's method. This time, the starting value is $z = 1$, each step maps $z \mapsto z(1.5 - 0.5yz^2)$, and we perform a fixed number of steps. Our final result combines r with $y^{-1/2}$ as $x^{-1/2} = 10^{-2} r y^{-1/2}$.

First normalize the input, then check the parity of the exponent #1. If it is even, the result's exponent will be $-#1/2$, otherwise it will be $(#1 - 1)/2$ (except in the case where the input was an exact power of 100). The `auxii` function receives as #1 the result's exponent just computed, as #2 the starting value for the iteration giving r (the values 168 and 535 lead to the least number of iterations before convergence, on average), as #3 and #4 one empty argument and one 0, depending on the parity of the original exponent, as #5 and #6 the normalized mantissa (#5 $\in [1000, 9999]$), and as #7 the continuation. It sets up the iteration giving r : the `esti` function thus receives the initial two guesses #2 and 0, an approximation #5 of $10^4 x$ (its first block of digits), and the empty/zero arguments #3 and #4, followed by the mantissa and an altered continuation where we have stored the result's exponent.

```

15430 \cs_new:Npn \__fp_ep_isqrt:wwn #1,#2;
15431 {
15432   \__fp_ep_to_ep:wwN #1,#2;
15433   \__fp_ep_isqrt_auxi:wwn
15434 }
15435 \cs_new:Npn \__fp_ep_isqrt_auxi:wwn #1,
15436 {
15437   \exp_after:wN \__fp_ep_isqrt_auxii:wwnnwn
15438   \int_value:w \__fp_int_eval:w
15439   \int_if_odd:nTF {#1}
15440     { (1 - #1) / 2 , 535 , { 0 } { } }
15441     { 1 - #1 / 2 , 168 , { } { 0 } }
15442 }
15443 \cs_new:Npn \__fp_ep_isqrt_auxii:wwnnwn #1, #2, #3#4 #5#6; #7
15444 {
15445   \__fp_ep_isqrt_esti:wwnnwn #2, 0, #5, {#3} {#4}
15446   {#5} #6 ; { #7 #1 , }
15447 }
```

(End definition for `__fp_ep_isqrt:wwn`, `__fp_ep_isqrt_aux:wwn`, and `__fp_ep_isqrt_auxii:wwnnwn`.)

If the last two approximations gave the same result, we are done: call the `estii` function to clean up. Otherwise, evaluate $(\langle prev \rangle + 1.005 \cdot 10^8 \text{ or } 9 / (\langle prev \rangle \cdot x))/2$, as the next approximation: omitting the 1.005 factor, this would be Newton's method. We can check by brute force that if #4 is empty (the original exponent was even), the process computes an integer slightly larger than $100/\sqrt{x}$, while if #4 is 0 (the original exponent was odd), the result is an integer slightly larger than $100/\sqrt{x/10}$. Once we are done, we evaluate $100r^2/2$ or $10r^2/2$ (when the exponent is even or odd, respectively) and feed that to `estiii`. This third auxiliary finds $y_{\text{even}}/2 = 10^{-4} r^2 x/2$ or $y_{\text{odd}}/2 = 10^{-5} r^2 x/2$

(again, depending on earlier parity). A simple program shows that $y \in [1, 1.0201]$. The number $y/2$ is fed to `__fp_ep_isqrt_epsilon:wN`, which computes $1/\sqrt{y}$, and we finally multiply the result by r .

```

15448 \cs_new:Npn \__fp_ep_isqrt_esti:wwnnwn #1, #2, #3, #4
15449 {
15450   \if_int_compare:w #1 = #2 \exp_stop_f:
15451     \exp_after:wN \__fp_ep_isqrt_estii:wwnnwn
15452   \fi:
15453   \exp_after:wN \__fp_ep_isqrt_esti:wwnnwn
15454   \int_value:w \__fp_int_eval:w
15455     (#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
15456   #1, #3, {#4}
15457 }
15458 \cs_new:Npn \__fp_ep_isqrt_estii:wwnnwn #1, #2, #3, #4#5
15459 {
15460   \exp_after:wN \__fp_ep_isqrt_estiii:NNNNNwwn
15461   \int_value:w \__fp_int_eval:w 1000 0000 + #2 * #2 #5 * 5
15462   \exp_after:wN , \int_value:w \__fp_int_eval:w 10000 + #2 ;
15463 }
15464 \cs_new:Npn \__fp_ep_isqrt_estiii:NNNNNwwn 1#1#2#3#4#5#6, 1#7#8; #9;
15465 {
15466   \__fp_fixed_mul_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
15467   \__fp_ep_isqrt_epsilon:wN
15468   \__fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
15469 }

```

(End definition for `__fp_ep_isqrt_esti:wwnnwn`, `__fp_ep_isqrt_estii:wwnnwn`, and `__fp_ep_isqrt_estiii:NNNNNwwn`.)

`__fp_ep_isqrt_epsilon:wN`
`__fp_ep_isqrt_epsilonii:wwN`

Here, we receive a fixed point number $y/2$ with $y \in [1, 1.0201]$. Starting from $z = 1$ we iterate $z \mapsto z(3/2 - z^2 y/2)$. In fact, we start from the first iteration $z = 3/2 - y/2$ to avoid useless multiplications. The `epsii` auxiliary receives z as `#1` and y as `#2`.

```

15470 \cs_new:Npn \__fp_ep_isqrt_epsilon:wN #1;
15471 {
15472   \__fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}; #1;
15473   \__fp_ep_isqrt_epsilonii:wwN #1;
15474   \__fp_ep_isqrt_epsilonii:wwN #1;
15475   \__fp_ep_isqrt_epsilonii:wwN #1;
15476 }
15477 \cs_new:Npn \__fp_ep_isqrt_epsilonii:wwN #1; #2;
15478 {
15479   \__fp_fixed_mul:wwn #1; #1;
15480   \__fp_fixed_mul_sub_back:wwn #2;
15481     {15000}{0000}{0000}{0000}{0000}{0000};
15482   \__fp_fixed_mul:wwn #1;
15483 }

```

(End definition for `__fp_ep_isqrt_epsilon:wN` and `__fp_ep_isqrt_epsilonii:wwN`.)

29.11 Converting from fixed point to floating point

After computing Taylor series, we wish to convert the result from extended precision (with or without an exponent) to the public floating point format. The functions here

should be called within an integer expression for the overall exponent of the floating point.

`__fp_ep_to_float_o:wwN` An extended-precision number is simply a comma-delimited exponent followed by a fixed point number. Leave the exponent in the current integer expression then convert the fixed point number.

```

15484 \cs_new:Npn \__fp_ep_to_float_o:wwN #1,
15485   { + \__fp_int_eval:w #1 \__fp_fixed_to_float_o:wN }
15486 \cs_new:Npn \__fp_ep_inv_to_float_o:wwN #1,#2;
15487   {
15488     \__fp_ep_div:wwwN 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
15489     \__fp_ep_to_float_o:wwN
15490   }

```

(End definition for `__fp_ep_to_float_o:wwN` and `__fp_ep_inv_to_float_o:wwN`.)

`__fp_fixed_inv_to_float_o:wN` Another function which reduces to converting an extended precision number to a float.

```

15491 \cs_new:Npn \__fp_fixed_inv_to_float_o:wN
15492   { \__fp_ep_inv_to_float_o:wwN 0, }

```

(End definition for `__fp_fixed_inv_to_float_o:wN`.)

`__fp_fixed_to_float_rad_o:wN` Converts the fixed point number #1 from degrees to radians then to a floating point number. This could perhaps remain in `l3fp-trig`.

```

15493 \cs_new:Npn \__fp_fixed_to_float_rad_o:wN #1;
15494   {
15495     \__fp_fixed_mul:wwN #1; {5729}{5779}{5130}{8232}{0876}{7981};
15496     { \__fp_ep_to_float_o:wwN 2, }
15497   }

```

(End definition for `__fp_fixed_to_float_rad_o:wN`.)

`__fp_fixed_to_float_o:wN` ... `__fp_int_eval:w` $\langle exponent \rangle$ `__fp_fixed_to_float_o:wN` $\{\langle a_1 \rangle\} \{\langle a_2 \rangle\} \{\langle a_3 \rangle\}$
`__fp_fixed_to_float_o:Nw` $\{\langle a_4 \rangle\} \{\langle a_5 \rangle\} \{\langle a_6 \rangle\}$; $\langle sign \rangle$
yields
 $\langle exponent' \rangle$; $\{\langle a'_1 \rangle\} \{\langle a'_2 \rangle\} \{\langle a'_3 \rangle\} \{\langle a'_4 \rangle\}$;

And the `to_fixed` version gives six brace groups instead of 4, ensuring that $1000 \leq \langle a'_1 \rangle \leq 9999$. At this stage, we know that $\langle a_1 \rangle$ is positive (otherwise, it is sign of an error before), and we assume that it is less than 10^8 .⁹

```

15498 \cs_new:Npn \__fp_fixed_to_float_o:Nw #1#2;
15499   { \__fp_fixed_to_float_o:wN #2; #1 }
15500 \cs_new:Npn \__fp_fixed_to_float_o:wN #1#2#3#4#5#6; #7
15501   { % for the 8-digit-at-the-start thing
15502     + \__fp_int_eval:w \c__fp_block_int
15503     \exp_after:wN \exp_after:wN
15504     \exp_after:wN \__fp_fixed_to_loop:N
15505     \exp_after:wN \use_none:n
15506     \int_value:w \__fp_int_eval:w
15507     1 0000 0000 + #1 \exp_after:wN \__fp_use_none_stop_f:n
15508     \int_value:w 1#2 \exp_after:wN \__fp_use_none_stop_f:n
15509     \int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n

```

⁹Bruno: I must double check this assumption.

```

15510     \int_value:w 1#5#6
15511     \exp_after:wN ;
15512     \exp_after:wN ;
15513   }
15514 \cs_new:Npn \__fp_fixed_to_loop:N #1
15515 {
15516   \if_meaning:w 0 #1
15517     - 1
15518     \exp_after:wN \__fp_fixed_to_loop:N
15519   \else:
15520     \exp_after:wN \__fp_fixed_to_loop_end:w
15521     \exp_after:wN #1
15522   \fi:
15523 }
15524 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
15525 {
15526   \if_meaning:w ; #1
15527     \exp_after:wN \__fp_fixed_to_float_zero:w
15528   \else:
15529     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
15530     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
15531     \exp_after:wN \__fp_fixed_to_float_pack:ww
15532     \exp_after:wN ;
15533   \fi:
15534   #1 #2 0000 0000 0000 0000 ;
15535 }
15536 \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
15537 {
15538   - 2 * \c__fp_max_exponent_int ;
15539   {0000} {0000} {0000} {0000} ;
15540 }
15541 \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
15542 {
15543   \if_int_compare:w #2 > 4 \exp_stop_f:
15544     \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
15545   \fi:
15546   ; #1 ;
15547 }
15548 \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
15549 {
15550   \exp_after:wN \__fp_basics_pack_high:NNNNNw
15551   \int_value:w \__fp_int_eval:w 1 #1#2
15552   \exp_after:wN \__fp_basics_pack_low:NNNNNw
15553   \int_value:w \__fp_int_eval:w 1 #3#4 + 1 ;
15554 }

```

(End definition for __fp_fixed_to_float_o:wN and __fp_fixed_to_float_o:Nw.)

```

15555 </initex | package>

```

30 l3fp-expo implementation

```

15556 <*initex | package>
15557 <@@=fp>

```

```

\__fp_parse_word_exp:N Unary functions.
\__fp_parse_word_ln:N
15558 \cs_new:Npn \__fp_parse_word_exp:N
15559 { \__fp_parse_unary_function:NNN \__fp_exp_o:w ? }
15560 \cs_new:Npn \__fp_parse_word_ln:N
15561 { \__fp_parse_unary_function:NNN \__fp_ln_o:w ? }

(End definition for \__fp_parse_word_exp:N and \__fp_parse_word_ln:N.)

```

30.1 Logarithm

30.1.1 Work plan

As for many other functions, we filter out special cases in `__fp_ln_o:w`. Then `__fp_ln_npos_o:w` receives a positive normal number, which we write in the form $a \cdot 10^b$ with $a \in [0.1, 1)$.

The rest of this section is actually not in sync with the code. Or is the code not in sync with the section? In the current code, $c \in [1, 10]$ is such that $0.7 \leq ac < 1.4$.

We are given a positive normal number, of the form $a \cdot 10^b$ with $a \in [0.1, 1)$. To compute its logarithm, we find a small integer $5 \leq c < 50$ such that $0.91 \leq ac/5 < 1.1$, and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms $\ln(10)$ and $\ln(c/5)$ are looked up in a table. The last term is computed using the following Taylor series of \ln near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + t^2 \left(\frac{1}{3} + t^2 \left(\frac{1}{5} + t^2 \left(\frac{1}{7} + t^2 \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

where $t = 1 - 10/(ac + 5)$. We can now see one reason for the choice of $ac \sim 5$: then $ac + 5 = 10(1 - \epsilon)$ with $-0.05 < \epsilon \leq 0.045$, hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4) \dots,$$

is not too difficult to compute.

30.1.2 Some constants

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don't need the value of $\ln(5)$.

```

\c__fp_ln_i_fixed_tl
\c__fp_ln_ii_fixed_tl
\c__fp_ln_iii_fixed_tl
\c__fp_ln_iv_fixed_tl
\c__fp_ln_vi_fixed_tl
\c__fp_ln_vii_fixed_tl
\c__fp_ln_viii_fixed_tl
\c__fp_ln_ix_fixed_tl
\c__fp_ln_x_fixed_tl
15562 \tl_const:Nn \c__fp_ln_i_fixed_tl { {0000}{0000}{0000}{0000}{0000}{0000}; }
15563 \tl_const:Nn \c__fp_ln_ii_fixed_tl { {6931}{4718}{0559}{9453}{0941}{7232}; }
15564 \tl_const:Nn \c__fp_ln_iii_fixed_tl { {10986}{1228}{8668}{1096}{9139}{5245}; }
15565 \tl_const:Nn \c__fp_ln_iv_fixed_tl { {13862}{9436}{1119}{8906}{1883}{4464}; }
15566 \tl_const:Nn \c__fp_ln_vi_fixed_tl { {17917}{5946}{9228}{0550}{0081}{2477}; }
15567 \tl_const:Nn \c__fp_ln_vii_fixed_tl { {19459}{1014}{9055}{3133}{0510}{5353}; }
15568 \tl_const:Nn \c__fp_ln_viii_fixed_tl { {20794}{4154}{1679}{8359}{2825}{1696}; }
15569 \tl_const:Nn \c__fp_ln_ix_fixed_tl { {21972}{2457}{7336}{2193}{8279}{0490}; }
15570 \tl_const:Nn \c__fp_ln_x_fixed_tl { {23025}{8509}{2994}{0456}{8401}{7991}; }

```

(End definition for `\c__fp_ln_i_fixed_tl` and others.)

30.1.3 Sign, exponent, and special numbers

`__fp_ln_o:w` The logarithm of negative numbers (including $-\infty$ and -0) raises the “invalid” exception. The logarithm of $+0$ is $-\infty$, raising a division by zero exception. The logarithm of $+\infty$ or a `nan` is itself. Positive normal numbers call `__fp_ln_npos_o:w`.

```

15571 \cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15572 {
15573   \if_meaning:w 2 #3
15574     \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
15575   \fi:
15576   \if_case:w #2 \exp_stop_f:
15577     \__fp_case_use:nw
15578     { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
15579   \or:
15580   \else:
15581     \__fp_case_return_same_o:w
15582   \fi:
15583   \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
15584 }

```

(End definition for `__fp_ln_o:w`.)

30.1.4 Absolute ln

`__fp_ln_npos_o:w` We catch the case of a significand very close to 0.1 or to 1. In all other cases, the final result is at least 10^{-4} , and then an error of $0.5 \cdot 10^{-20}$ is acceptable.

```

15585 \cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
15586 { %%^A todo: ln(1) should be "exact zero", not "underflow"
15587   \exp_after:wN \__fp_sanitizew
15588   \int_value:w % for the overall sign
15589   \if_int_compare:w #1 < 1 \exp_stop_f:
15590     2
15591   \else:
15592     0
15593   \fi:
15594   \exp_after:wN \exp_stop_f:
15595   \int_value:w \__fp_int_eval:w % for the exponent
15596   \__fp_ln_significand:NNNNnnnnN #2#3
15597   \__fp_ln_exponent:wn {#1}
15598 }

```

(End definition for `__fp_ln_npos_o:w`.)

`__fp_ln_significand:NNNNnnnnN` `__fp_ln_significand:NNNNnnnnN` $\langle X_1 \rangle$ $\{\langle X_2 \rangle\}$ $\{\langle X_3 \rangle\}$ $\{\langle X_4 \rangle\}$ $\langle continuation \rangle$
 This function expands to

$\langle continuation \rangle$ $\{\langle Y_1 \rangle\}$ $\{\langle Y_2 \rangle\}$ $\{\langle Y_3 \rangle\}$ $\{\langle Y_4 \rangle\}$ $\{\langle Y_5 \rangle\}$ $\{\langle Y_6 \rangle\}$;

where $Y = -\ln(X)$ as an extended fixed point.

```

15599 \cs_new:Npn \__fp_ln_significand:NNNNnnnnN #1#2#3#4
15600 {
15601   \exp_after:wN \__fp_ln_x_ii:wnnnn
15602   \int_value:w
15603   \if_case:w #1 \exp_stop_f:

```



```

15604     \or:
15605     \if_int_compare:w #2 < 4 \exp_stop_f:
15606     \__fp_int_eval:w 10 - #2
15607     \else:
15608     6
15609     \fi:
15610     \or: 4
15611     \or: 3
15612     \or: 2
15613     \or: 2
15614     \or: 2
15615     \else: 1
15616     \fi:
15617 ; { #1 #2 #3 #4 }
15618 }

```

(End definition for `__fp_ln_significand:NNNNnnnnN`.)

`__fp_ln_x_ii:wnnnn` We have thus found $c \in [1, 10]$ such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```

15619 \cs_new:Npn \__fp_ln_x_ii:wnnnn #1; #2#3#4#5
15620 {
15621     \exp_after:wN \__fp_ln_div_after:Nw
15622     \cs:w c__fp_ln_ \__fp_int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
15623     \int_value:w
15624     \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
15625     \int_value:w \__fp_int_eval:w
15626     \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
15627     \int_value:w \__fp_int_eval:w 9999 9990 + #1*#2#3 +
15628     \exp_after:wN \__fp_ln_x_iii:NNNNNw
15629     \int_value:w \__fp_int_eval:w 10 0000 0000 + #1*#4#5 ;
15630     {20000} {0000} {0000} {0000}
15631 } %^A todo: reoptimize (a generalization attempt failed).
15632 \cs_new:Npn \__fp_ln_x_iii:NNNNNw #1#2 #3#4#5#6 #7;
15633 { #1#2; {#3#4#5#6} {#7} }
15634 \cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
15635 {
15636     #1#2#3#4#5 + 1 ;
15637     {#1#2#3#4#5} {#6}
15638 }

```

The Taylor series to be used is expressed in terms of $t = (x - 1)/(x + 1) = 1 - 2/(x + 1)$. We now compute the quotient with extended precision, reusing some code from `__fp_-/_o:ww`. Note that $1 + x$ is known exactly.

To reuse notations from `l3fp-basics`, we want to compute A/Z with $A = 2$ and $Z = x + 1$. In `l3fp-basics`, we considered the case where both A and Z are arbitrary, in the range $[0.1, 1)$, and we had to monitor the growth of the sequence of remainders A , B , C , etc. to ensure that no overflow occurred during the computation of the next quotient. The main source of risk was our choice to define the quotient as roughly $10^9 \cdot A/10^5 \cdot Z$: then A was bound to be below $2.147 \dots$, and this limit was never far.

In our case, we can simply work with $10^8 \cdot A$ and $10^4 \cdot Z$, because our reason to work with higher powers has gone: we needed the integer $y \simeq 10^5 \cdot Z$ to be at least 10^4 , and now, the definition $y \simeq 10^4 \cdot Z$ suffices.

Let us thus define $y = \lfloor 10^4 \cdot Z \rfloor + 1 \in (1.7 \cdot 10^4, 2.4 \cdot 10^4]$, and

$$Q_1 = \left\lfloor \frac{\lfloor 10^8 \cdot A \rfloor}{y} - \frac{1}{2} \right\rfloor.$$

(The $1/2$ comes from how ε -TeX rounds.) As for division, it is easy to see that $Q_1 \leq 10^4 A/Z$, *i.e.*, Q_1 is an underestimate.

Exactly as we did for division, we set $B = 10^4 A - Q_1 Z$. Then

$$\begin{aligned} 10^4 B &\leq A_1 A_2 \cdot A_3 A_4 - \left(\frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \\ &\leq A_1 A_2 \left(1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \\ &\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y \end{aligned}$$

In the same way, and using $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$, and convexity, we get

$$\begin{aligned} 10^4 A &= 2 \cdot 10^4 \\ 10^4 B &\leq 10^8 \frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4 \\ 10^4 C &\leq 10^8 \frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4 \\ 10^4 D &\leq 10^8 \frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4 \\ 10^4 E &\leq 10^8 \frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4 \\ 10^4 F &\leq 10^8 \frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4 \end{aligned}$$

Note that we compute more steps than for division: since t is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don't need an exact rounding for transcendental functions, but just a faithful rounding).

`__fp_ln_x_iv:wnnnnnnnn <1 or 2> <8d> ; {\<4d>} {\<4d>} <fixed-tl>`

The number is x . Compute y by adding 1 to the five first digits.

```

15639 \cs_new:Npn \__fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
15640 {
15641   \exp_after:wN \__fp_div_significand_pack:NNN
15642   \int_value:w \__fp_int_eval:w
15643   \__fp_ln_div_i:w #1 ;
15644   #6 #7 ; {\#8} {\#9}
15645   {\#2} {\#3} {\#4} {\#5}
15646   { \exp_after:wN \__fp_ln_div_ii:wnn \int_value:w #1 }
15647   { \exp_after:wN \__fp_ln_div_ii:wnn \int_value:w #1 }
15648   { \exp_after:wN \__fp_ln_div_ii:wnn \int_value:w #1 }
15649   { \exp_after:wN \__fp_ln_div_ii:wnn \int_value:w #1 }
15650   { \exp_after:wN \__fp_ln_div_vi:wnn \int_value:w #1 }

```

```

15651 }
15652 \cs_new:Npn \__fp_ln_div_i:w #1;
15653 {
15654   \exp_after:wN \__fp_div_significand_calc:wwnnnnnnnn
15655   \int_value:w \__fp_int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
15656 }
15657 \cs_new:Npn \__fp_ln_div_ii:wwn #1; #2;#3 % y; B1;B2 <- for k=1
15658 {
15659   \exp_after:wN \__fp_div_significand_pack:NNN
15660   \int_value:w \__fp_int_eval:w
15661   \exp_after:wN \__fp_div_significand_calc:wwnnnnnnnn
15662   \int_value:w \__fp_int_eval:w 999999 + #2 #3 / #1 ; % Q2
15663   #2 #3 ;
15664 }
15665 \cs_new:Npn \__fp_ln_div_vi:wwn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
15666 {
15667   \exp_after:wN \__fp_div_significand_pack:NNN
15668   \int_value:w \__fp_int_eval:w 1000000 + #2 #3 / #1 ; % Q6
15669 }

```

We now have essentially

```

\__fp_ln_div_after:Nw <fixed t1>
\__fp_div_significand_pack:NNN 106 + Q1
\__fp_div_significand_pack:NNN 106 + Q2
\__fp_div_significand_pack:NNN 106 + Q3
\__fp_div_significand_pack:NNN 106 + Q4
\__fp_div_significand_pack:NNN 106 + Q5
\__fp_div_significand_pack:NNN 106 + Q6 ;
<exponent> ; <continuation>

```

where $\langle \text{fixed } t1 \rangle$ holds the logarithm of a number in $[1, 10]$, and $\langle \text{exponent} \rangle$ is the exponent. Also, the expansion is done backwards. Then `__fp_div_significand_pack:NNN` puts things in the correct order to add the Q_i together and put semicolons between each piece. Once those have been expanded, we get

```

\__fp_ln_div_after:Nw <fixed-t1> <1d> ; <4d> ; <4d> ;
<4d> ; <4d> ; <4d> ; <4d> ; <exponent> ;

```

Just as with division, we know that the first two digits are 1 and 0 because of bounds on the final result of the division $2/(x+1)$, which is between roughly 0.8 and 1.2. We then compute $1 - 2/(x+1)$, after testing whether $2/(x+1)$ is greater than or smaller than 1.

```

15670 \cs_new:Npn \__fp_ln_div_after:Nw #1#2;
15671 {
15672   \if_meaning:w 0 #2
15673   \exp_after:wN \__fp_ln_t_small:Nw
15674   \else:
15675   \exp_after:wN \__fp_ln_t_large:NNw
15676   \exp_after:wN -
15677   \fi:
15678   #1
15679 }
15680 \cs_new:Npn \__fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
15681 {
15682   \exp_after:wN \__fp_ln_t_large:NNw

```

```

15683 \exp_after:wN + % <sign>
15684 \exp_after:wN #1
15685 \int_value:w \_fp_int_eval:w 9999 - #2 \exp_after:wN ;
15686 \int_value:w \_fp_int_eval:w 9999 - #3 \exp_after:wN ;
15687 \int_value:w \_fp_int_eval:w 9999 - #4 \exp_after:wN ;
15688 \int_value:w \_fp_int_eval:w 9999 - #5 \exp_after:wN ;
15689 \int_value:w \_fp_int_eval:w 9999 - #6 \exp_after:wN ;
15690 \int_value:w \_fp_int_eval:w 1 0000 - #7 ;
15691 }

```

```

\_fp_ln_t_large:NNw <sign> <fixed t1>
<t1>; <t2>; <t3>; <t4>; <t5>; <t6>;
<exponent>; <continuation>

```

Compute the square t^2 , and keep t at the end with its sign. We know that $t < 0.1765$, so every piece has at most 4 digits. However, since we were not careful in `_fp_ln_t_small:w`, they can have less than 4 digits.

```

15692 \cs_new:Npn \_fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
15693 {
15694   \exp_after:wN \_fp_ln_square_t_after:w
15695   \int_value:w \_fp_int_eval:w 9999 0000 + #3*#3
15696   \exp_after:wN \_fp_ln_square_t_pack:NNNNNw
15697   \int_value:w \_fp_int_eval:w 9999 0000 + 2*#3*#4
15698   \exp_after:wN \_fp_ln_square_t_pack:NNNNNw
15699   \int_value:w \_fp_int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
15700   \exp_after:wN \_fp_ln_square_t_pack:NNNNNw
15701   \int_value:w \_fp_int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
15702   \exp_after:wN \_fp_ln_square_t_pack:NNNNNw
15703   \int_value:w \_fp_int_eval:w
15704     1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
15705     + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
15706   % ; ; ;
15707   \exp_after:wN \_fp_ln_twice_t_after:w
15708   \int_value:w \_fp_int_eval:w -1 + 2*#3
15709   \exp_after:wN \_fp_ln_twice_t_pack:Nw
15710   \int_value:w \_fp_int_eval:w 9999 + 2*#4
15711   \exp_after:wN \_fp_ln_twice_t_pack:Nw
15712   \int_value:w \_fp_int_eval:w 9999 + 2*#5
15713   \exp_after:wN \_fp_ln_twice_t_pack:Nw
15714   \int_value:w \_fp_int_eval:w 9999 + 2*#6
15715   \exp_after:wN \_fp_ln_twice_t_pack:Nw
15716   \int_value:w \_fp_int_eval:w 9999 + 2*#7
15717   \exp_after:wN \_fp_ln_twice_t_pack:Nw
15718   \int_value:w \_fp_int_eval:w 10000 + 2*#8 ; ;
15719   { \_fp_ln_c:NwNw #1 }
15720   #2
15721 }
15722 \cs_new:Npn \_fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
15723 \cs_new:Npn \_fp_ln_twice_t_after:w #1; { ; ; ; {#1} }
15724 \cs_new:Npn \_fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
15725   { + #1#2#3#4#5 ; {#6} }
15726 \cs_new:Npn \_fp_ln_square_t_after:w 1 0 #1#2#3 #4;
15727   { \_fp_ln_Taylor:wwNw {0#1#2#3} {#4} }

```

(End definition for `_fp_ln_x_ii:wnnnn`.)

_fp_ln_Taylor:wwNw Denoting $T = t^2$, we get

```
\_fp_ln_Taylor:wwNw
{⟨T1⟩} {⟨T2⟩} {⟨T3⟩} {⟨T4⟩} {⟨T5⟩} {⟨T6⟩} ; ;
{⟨(2t)1⟩} {⟨(2t)2⟩} {⟨(2t)3⟩} {⟨(2t)4⟩} {⟨(2t)5⟩} {⟨(2t)6⟩} ;
{ \_fp_ln_c:NwNw ⟨sign⟩ }
⟨fixed t1⟩ ⟨exponent⟩ ; ⟨continuation⟩
```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + T \left(\frac{1}{3} + T \left(\frac{1}{5} + T \left(\frac{1}{7} + T \left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

The process looks as follows

```
\loop 5; A;
\div_int 5; 1.0; \add A; \mul T; {\loop \eval 5-2;}
\add 0.2; A; \mul T; {\loop \eval 5-2;}
\mul B; T; {\loop 3;}
\loop 3; C;
```

This uses the routine for dividing a number by a small integer ($< 10^4$).

```
15728 \cs_new:Npn \_fp_ln_Taylor:wwNw
15729 { \_fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000} ; }
15730 \cs_new:Npn \_fp_ln_Taylor_loop:www #1; #2; #3;
15731 {
15732   \if_int_compare:w #1 = 1 \exp_stop_f:
15733   \_fp_ln_Taylor_break:w
15734   \fi:
15735   \exp_after:wN \_fp_fixed_div_int:wwN \c\_fp_one_fixed_t1 #1;
15736   \_fp_fixed_add:wwn #2;
15737   \_fp_fixed_mul:wwn #3;
15738   {
15739     \exp_after:wN \_fp_ln_Taylor_loop:www
15740     \int_value:w \_fp_int_eval:w #1 - 2 ;
15741   }
15742   #3;
15743 }
15744 \cs_new:Npn \_fp_ln_Taylor_break:w \fi: #1 \_fp_fixed_add:wwn #2#3; #4 ;;
15745 {
15746   \fi:
15747   \exp_after:wN \_fp_fixed_mul:wwn
15748   \exp_after:wN { \int_value:w \_fp_int_eval:w 10000 + #2 } #3;
15749 }
```

(End definition for _fp_ln_Taylor:wwNw.)

```
\_fp_ln_c:NwNw ⟨sign⟩
{⟨r1⟩} {⟨r2⟩} {⟨r3⟩} {⟨r4⟩} {⟨r5⟩} {⟨r6⟩} ;
⟨fixed t1⟩ ⟨exponent⟩ ; ⟨continuation⟩
```

We are now reduced to finding $\ln(c)$ and $\langle exponent \rangle \ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get $\mathbf{b} \ln(10)$ and add or subtract.

For now, $\ln(x)$ is given as $\cdot 10^0$. Unless both the exponent is 1 and $c = 1$, we shift to working in units of $\cdot 10^4$, since the final result is at least $\ln(10/7) \simeq 0.35$.

```

15750 \cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
15751 {
15752   \if_meaning:w + #1
15753     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wnw
15754   \else:
15755     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wnw
15756   \fi:
15757   #3 #2 ;
15758 }

```

(End definition for $\backslash_fp_ln_c:NwNw$.)

```

\__fp_ln_exponent:wn
    \__fp_ln_exponent:wn
    {\langle s_1 \rangle} {\langle s_2 \rangle} {\langle s_3 \rangle} {\langle s_4 \rangle} {\langle s_5 \rangle} {\langle s_6 \rangle} ;
    {\langle exponent \rangle}

```

Compute $\langle exponent \rangle$ times $\ln(10)$. Apart from the cases where $\langle exponent \rangle$ is 0 or 1, the result is necessarily at least $\ln(10) \simeq 2.3$ in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order 10^4 . Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

```

15759 \cs_new:Npn \__fp_ln_exponent:wn #1; #2
15760 {
15761   \if_case:w #2 \exp_stop_f:
15762     0 \__fp_case_return:nw { \__fp_fixed_to_float_o:Nw 2 }
15763   \or:
15764     \exp_after:wN \__fp_ln_exponent_one:ww \int_value:w
15765   \else:
15766     \if_int_compare:w #2 > 0 \exp_stop_f:
15767       \exp_after:wN \__fp_ln_exponent_small:NNww
15768       \exp_after:wN 0
15769       \exp_after:wN \__fp_fixed_sub:wnw \int_value:w
15770     \else:
15771       \exp_after:wN \__fp_ln_exponent_small:NNww
15772       \exp_after:wN 2
15773       \exp_after:wN \__fp_fixed_add:wnw \int_value:w -
15774     \fi:
15775   \fi:
15776   #2; #1;
15777 }

```

Now we painfully write all the cases.¹⁰ No overflow nor underflow can happen, except when computing $\ln(1)$.

```

15778 \cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
15779 {
15780   0
15781   \exp_after:wN \__fp_fixed_sub:wnw \c__fp_ln_x_fixed_tl #1;
15782   \__fp_fixed_to_float_o:wN 0
15783 }

```

¹⁰Bruno: do rounding.

For small exponents, we just drop one block of digits, and set the exponent of the log to 4 (minus any shift coming from leading zeros in the conversion from fixed point to floating point). Note that here the exponent has been made positive.

```

15784 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
15785 {
15786     4
15787     \exp_after:wN \__fp_fixed_mul:wwn
15788     \c__fp_ln_x_fixed_tl
15789     {#3}{0000}{0000}{0000}{0000}{0000} ;
15790     #2
15791     {0000}{#4}{#5}{#6}{#7}{#8};
15792     \__fp_fixed_to_float_o:wN #1
15793 }

```

(End definition for __fp_ln_exponent:wn.)

30.2 Exponential

30.2.1 Sign, exponent, and special numbers

__fp_exp_o:w

```

15794 \cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15795 {
15796     \if_case:w #2 \exp_stop_f:
15797     \__fp_case_return_o:Nw \c_one_fp
15798     \or:
15799     \exp_after:wN \__fp_exp_normal_o:w
15800     \or:
15801     \if_meaning:w 0 #3
15802     \exp_after:wN \__fp_case_return_o:Nw
15803     \exp_after:wN \c_inf_fp
15804     \else:
15805     \exp_after:wN \__fp_case_return_o:Nw
15806     \exp_after:wN \c_zero_fp
15807     \fi:
15808     \or:
15809     \__fp_case_return_same_o:w
15810     \fi:
15811     \s__fp \__fp_chk:w #2#3#4;
15812 }

```

(End definition for __fp_exp_o:w.)

__fp_exp_normal_o:w

__fp_exp_pos_o:NNwnw

__fp_exp_overflow:NN

```

15813 \cs_new:Npn \__fp_exp_normal_o:w \s__fp \__fp_chk:w 1#1
15814 {
15815     \if_meaning:w 0 #1
15816     \__fp_exp_pos_o:NNwnw + \__fp_fixed_to_float_o:wN
15817     \else:
15818     \__fp_exp_pos_o:NNwnw - \__fp_fixed_inv_to_float_o:wN
15819     \fi:
15820 }
15821 \cs_new:Npn \__fp_exp_pos_o:NNwnw #1#2#3 \fi: #4#5;
15822 {

```

```

15823 \fi:
15824 \if_int_compare:w #4 > \c__fp_max_exp_exponent_int
15825 \token_if_eq_charcode:NNTF + #1
15826 { \__fp_exp_overflow:NN \__fp_overflow:w \c_inf_fp }
15827 { \__fp_exp_overflow:NN \__fp_underflow:w \c_zero_fp }
15828 \exp:w
15829 \else:
15830 \exp_after:wN \__fp_sanitize:Nw
15831 \exp_after:wN 0
15832 \int_value:w #1 \__fp_int_eval:w
15833 \if_int_compare:w #4 < 0 \exp_stop_f:
15834 \exp_after:wN \use_i:nn
15835 \else:
15836 \exp_after:wN \use_ii:nn
15837 \fi:
15838 {
15839 0
15840 \__fp_decimate:nNnnnn { - #4 }
15841 \__fp_exp_Taylor:Nnnwn
15842 }
15843 {
15844 \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 }
15845 \__fp_exp_pos_large:NnnNwn
15846 }
15847 #5
15848 {#4}
15849 #1 #2 0
15850 \exp:w
15851 \fi:
15852 \exp_after:wN \exp_end:
15853 }
15854 \cs_new:Npn \__fp_exp_overflow:NN #1#2
15855 {
15856 \exp_after:wN \exp_after:wN
15857 \exp_after:wN #1
15858 \exp_after:wN #2
15859 }

```

(End definition for __fp_exp_normal_o:w, __fp_exp_pos_o:Nnnwn, and __fp_exp_overflow:NN.)

```

\__fp_exp_Taylor:Nnnwn
\__fp_exp_Taylor_loop:www
\__fp_exp_Taylor_break:Nww

```

This function is called for numbers in the range $[10^{-9}, 10^{-1}]$. We compute 10 terms of the Taylor series. The first argument is irrelevant (rounding digit used by some other functions). The next three arguments, at least 16 digits, delimited by a semicolon, form a fixed point number, so we pack it in blocks of 4 digits.

```

15860 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
15861 {
15862 #6
15863 \__fp_pack_twice_four:wNNNNNNNN
15864 \__fp_pack_twice_four:wNNNNNNNN
15865 \__fp_pack_twice_four:wNNNNNNNN
15866 \__fp_exp_Taylor_ii:ww
15867 ; #2#3#4 0000 0000 ;
15868 }
15869 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;

```



```

15870 { \_fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s_stop }
15871 \cs_new:Npn \_fp_exp_Taylor_loop:www #1; #2; #3;
15872 {
15873   \if_int_compare:w #1 = 1 \exp_stop_f:
15874   \exp_after:wN \_fp_exp_Taylor_break:Nww
15875   \fi:
15876   \_fp_fixed_div_int:wwN #3 ; #1 ;
15877   \_fp_fixed_add_one:wN
15878   \_fp_fixed_mul:wwN #2 ;
15879   {
15880     \exp_after:wN \_fp_exp_Taylor_loop:www
15881     \int_value:w \_fp_int_eval:w #1 - 1 ;
15882     #2 ;
15883   }
15884 }
15885 \cs_new:Npn \_fp_exp_Taylor_break:Nww #1 #2; #3 \s_stop
15886 { \_fp_fixed_add_one:wN #2 ; }

```

(End definition for _fp_exp_Taylor:Nnnwn, _fp_exp_Taylor_loop:www, and _fp_exp_Taylor_break:Nww.)

```

\_fp_exp_pos_large:NnnNwn
\_fp_exp_large_after:wwN
  \_fp_exp_large:w
  \_fp_exp_large_v:wN
  \_fp_exp_large_iv:wN
  \_fp_exp_large_iii:wN
  \_fp_exp_large_ii:wN
  \_fp_exp_large_i:wN
  \_fp_exp_large_:wN

```

The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros). The third argument is the integer part of our number, then we have the decimal part delimited by a semicolon, and finally the exponent, in the range [0,5]. Remove leading zeros from the integer part: putting #4 in there too ensures that an integer part of 0 is also removed. Then read digits one by one, looking up $\exp(\langle digit \rangle \cdot 10^{\langle exponent \rangle})$ in a table, and multiplying that to the current total. The loop is done by having the auxiliary for one exponent call the auxiliary for the next exponent. The current total is expressed by leaving the exponent behind in the input stream (we are currently within an _fp_int_eval:w), and keeping track of a fixed point number, #1 for the numbered auxiliaries. Our usage of \if_case:w is somewhat dirty for optimization: T_EX jumps to the appropriate case, but we then close the \if_case:w “by hand”, using \or: and \fi: as delimiters.

```

15887 \cs_new:Npn \_fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
15888 {
15889   \exp_after:wN \exp_after:wN
15890   \cs:w \_fp_exp_large_ \_fp_int_to_roman:w #6 :wN \exp_after:wN \cs_end:
15891   \exp_after:wN \c_fp_one_fixed_tl
15892   \int_value:w #3 #4 \exp_stop_f:
15893   #5 00000 ;
15894 }
15895 \cs_new:Npn \_fp_exp_large:w #1 \or: #2 \fi:
15896 { \fi: \_fp_fixed_mul:wwN #1; }
15897 \cs_new:Npn \_fp_exp_large_v:wN #1; #2
15898 {
15899   \if_case:w #2 ~ \exp_after:wN \_fp_fixed_continue:wn \or:
15900   + 4343 \_fp_exp_large:w {8806}{8182}{2566}{2921}{5872}{6150} \or:
15901   + 8686 \_fp_exp_large:w {7756}{0047}{2598}{6861}{0458}{3204} \or:
15902   + 13029 \_fp_exp_large:w {6830}{5723}{7791}{4884}{1932}{7351} \or:
15903   + 17372 \_fp_exp_large:w {6015}{5609}{3095}{3052}{3494}{7574} \or:
15904   + 21715 \_fp_exp_large:w {5297}{7951}{6443}{0315}{3251}{3576} \or:
15905   + 26058 \_fp_exp_large:w {4665}{6719}{0099}{3379}{5527}{2929} \or:
15906   + 30401 \_fp_exp_large:w {4108}{9724}{3326}{3186}{5271}{5665} \or:

```

```

15907         + 34744 \__fp_exp_large:w {3618}{6973}{3140}{0875}{3856}{4102} \or:
15908         + 39087 \__fp_exp_large:w {3186}{9209}{6113}{3900}{6705}{9685} \or:
15909     \fi:
15910     #1;
15911     \__fp_exp_large_iv:wN
15912 }
15913 \cs_new:Npn \__fp_exp_large_iv:wN #1; #2
15914 {
15915     \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn \or:
15916     + 435 \__fp_exp_large:w {1970}{0711}{1401}{7046}{9938}{8888} \or:
15917     + 869 \__fp_exp_large:w {3881}{1801}{9428}{4368}{5764}{8232} \or:
15918     + 1303 \__fp_exp_large:w {7646}{2009}{8905}{4704}{8893}{1073} \or:
15919     + 1738 \__fp_exp_large:w {1506}{3559}{7005}{0524}{9009}{7592} \or:
15920     + 2172 \__fp_exp_large:w {2967}{6283}{8402}{3667}{0689}{6630} \or:
15921     + 2606 \__fp_exp_large:w {5846}{4389}{5650}{2114}{7278}{5046} \or:
15922     + 3041 \__fp_exp_large:w {1151}{7900}{5080}{6878}{2914}{4154} \or:
15923     + 3475 \__fp_exp_large:w {2269}{1083}{0850}{6857}{8724}{4002} \or:
15924     + 3909 \__fp_exp_large:w {4470}{3047}{3316}{5442}{6408}{6591} \or:
15925     \fi:
15926     #1;
15927     \__fp_exp_large_iii:wN
15928 }
15929 \cs_new:Npn \__fp_exp_large_iii:wN #1; #2
15930 {
15931     \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn \or:
15932     + 44 \__fp_exp_large:w {2688}{1171}{4181}{6135}{4484}{1263} \or:
15933     + 87 \__fp_exp_large:w {7225}{9737}{6812}{5749}{2581}{7748} \or:
15934     + 131 \__fp_exp_large:w {1942}{4263}{9524}{1255}{9365}{8421} \or:
15935     + 174 \__fp_exp_large:w {5221}{4696}{8976}{4143}{9505}{8876} \or:
15936     + 218 \__fp_exp_large:w {1403}{5922}{1785}{2837}{4107}{3977} \or:
15937     + 261 \__fp_exp_large:w {3773}{0203}{0092}{9939}{8234}{0143} \or:
15938     + 305 \__fp_exp_large:w {1014}{2320}{5473}{5004}{5094}{5533} \or:
15939     + 348 \__fp_exp_large:w {2726}{3745}{7211}{2566}{5673}{6478} \or:
15940     + 391 \__fp_exp_large:w {7328}{8142}{2230}{7421}{7051}{8866} \or:
15941     \fi:
15942     #1;
15943     \__fp_exp_large_ii:wN
15944 }
15945 \cs_new:Npn \__fp_exp_large_ii:wN #1; #2
15946 {
15947     \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn \or:
15948     + 5 \__fp_exp_large:w {2202}{6465}{7948}{0671}{6516}{9579} \or:
15949     + 9 \__fp_exp_large:w {4851}{6519}{5409}{7902}{7796}{9107} \or:
15950     + 14 \__fp_exp_large:w {1068}{6474}{5815}{2446}{2146}{9905} \or:
15951     + 18 \__fp_exp_large:w {2353}{8526}{6837}{0199}{8540}{7900} \or:
15952     + 22 \__fp_exp_large:w {5184}{7055}{2858}{7072}{4640}{8745} \or:
15953     + 27 \__fp_exp_large:w {1142}{0073}{8981}{5684}{2836}{6296} \or:
15954     + 31 \__fp_exp_large:w {2515}{4386}{7091}{9167}{0062}{6578} \or:
15955     + 35 \__fp_exp_large:w {5540}{6223}{8439}{3510}{0525}{7117} \or:
15956     + 40 \__fp_exp_large:w {1220}{4032}{9431}{7840}{8020}{0271} \or:
15957     \fi:
15958     #1;
15959     \__fp_exp_large_i:wN
15960 }

```

```

15961 \cs_new:Npn \__fp_exp_large_i:wn #1; #2
15962 {
15963   \if_case:w #2 ~      \exp_after:wn \__fp_fixed_continue:wn \or:
15964     + 1 \__fp_exp_large:w {2718}{2818}{2845}{9045}{2353}{6029} \or:
15965     + 1 \__fp_exp_large:w {7389}{0560}{9893}{0650}{2272}{3043} \or:
15966     + 2 \__fp_exp_large:w {2008}{5536}{9231}{8766}{7740}{9285} \or:
15967     + 2 \__fp_exp_large:w {5459}{8150}{0331}{4423}{9078}{1103} \or:
15968     + 3 \__fp_exp_large:w {1484}{1315}{9102}{5766}{0342}{1116} \or:
15969     + 3 \__fp_exp_large:w {4034}{2879}{3492}{7351}{2260}{8387} \or:
15970     + 4 \__fp_exp_large:w {1096}{6331}{5842}{8458}{5992}{6372} \or:
15971     + 4 \__fp_exp_large:w {2980}{9579}{8704}{1728}{2747}{4359} \or:
15972     + 4 \__fp_exp_large:w {8103}{0839}{2757}{5384}{0077}{1000} \or:
15973   \fi:
15974   #1;
15975   \__fp_exp_large_:wn
15976 }
15977 \cs_new:Npn \__fp_exp_large_:wn #1; #2
15978 {
15979   \if_case:w #2 ~      \exp_after:wn \__fp_fixed_continue:wn \or:
15980     + 1 \__fp_exp_large:w {1105}{1709}{1807}{5647}{6248}{1171} \or:
15981     + 1 \__fp_exp_large:w {1221}{4027}{5816}{0169}{8339}{2107} \or:
15982     + 1 \__fp_exp_large:w {1349}{8588}{0757}{6003}{1039}{8374} \or:
15983     + 1 \__fp_exp_large:w {1491}{8246}{9764}{1270}{3178}{2485} \or:
15984     + 1 \__fp_exp_large:w {1648}{7212}{7070}{0128}{1468}{4865} \or:
15985     + 1 \__fp_exp_large:w {1822}{1188}{0039}{0508}{9748}{7537} \or:
15986     + 1 \__fp_exp_large:w {2013}{7527}{0747}{0476}{5216}{2455} \or:
15987     + 1 \__fp_exp_large:w {2225}{5409}{2849}{2467}{6045}{7954} \or:
15988     + 1 \__fp_exp_large:w {2459}{6031}{1115}{6949}{6638}{0013} \or:
15989   \fi:
15990   #1;
15991   \__fp_exp_large_after:wnn
15992 }
15993 \cs_new:Npn \__fp_exp_large_after:wnn #1; #2; #3
15994 {
15995   \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
15996   \__fp_fixed_mul:wnn #1;
15997 }

```

(End definition for __fp_exp_pos_large:NnnNwn and others.)

30.3 Power

Raising a number a to a power b leads to many distinct situations.

a^b	$-\infty$	$(-\infty, -0)$	$-\text{integer}$	± 0	$+\text{integer}$	$(0, \infty)$	$+\infty$	NaN
$+\infty$	$+0$		$+0$	$+1$	$+\infty$	$+\infty$	$+\infty$	NaN
$(1, \infty)$	$+0$		$+ a ^b$	$+1$	$+ a ^b$	$+\infty$	$+\infty$	NaN
$+1$	$+1$		$+1$	$+1$	$+1$	$+1$	$+1$	$+1$
$(0, 1)$	$+\infty$		$+ a ^b$	$+1$	$+ a ^b$	$+0$	$+\infty$	NaN
$+0$	$+\infty$		$+\infty$	$+1$	$+0$	$+0$	$+\infty$	NaN
-0	$+\infty$	NaN	$(-1)^b \infty$	$+1$	$(-1)^b 0$	$+0$	$+0$	NaN
$(-1, 0)$	$+\infty$	NaN	$(-1)^b a ^b$	$+1$	$(-1)^b a ^b$	NaN	$+0$	NaN
-1	$+1$	NaN	$(-1)^b$	$+1$	$(-1)^b$	NaN	$+1$	NaN
$(-\infty, -1)$	$+0$	NaN	$(-1)^b a ^b$	$+1$	$(-1)^b a ^b$	NaN	$+\infty$	NaN
$-\infty$	$+0$	$+0$	$(-1)^b 0$	$+1$	$(-1)^b \infty$	NaN	$+\infty$	NaN
NaN	NaN	NaN	NaN	$+1$	NaN	NaN	NaN	NaN

We distinguished in this table the cases of finite (positive or negative) integer exponents, as $(-1)^b$ is defined in that case. One peculiarity of this operation is that $\text{NaN}^0 = 1^{\text{NaN}} = 1$, because this relation is obeyed for any number, even $\pm\infty$.

`_fp_o:ww` We cram most of the tests into a single function to save csnames. First treat the case $b = 0$: $a^0 = 1$ for any a , even nan . Then test the sign of a .

- If it is positive, and a is a normal number, call `_fp_pow_normal_o:ww` followed by the two `fp` a and b . For $a = +0$ or $+\text{inf}$, call `_fp_pow_zero_or_inf:ww` instead, to return either $+0$ or $+\infty$ as appropriate.
- If a is a `nan`, then skip to the next semicolon (which happens to be conveniently the end of b) and return `nan`.
- Finally, if a is negative, compute a^b (`_fp_pow_normal_o:ww` which ignores the sign of its first operand), and keep an extra copy of a and b (the second brace group, containing $\{ b a \}$, is inserted between a and b). Then do some tests to find the final sign of the result if it exists.

```

15998 \cs_new:cpn { \_fp\_ \iow_char:N \^ \_o:ww }
15999   \s__fp \_fp_chk:w #1#2#3; \s__fp \_fp_chk:w #4#5#6;
16000   {
16001     \if_meaning:w 0 #4
16002       \_fp_case_return_o:Nw \c_one_fp
16003     \fi:
16004     \if_case:w #2 \exp_stop_f:
16005       \exp_after:wN \use_i:nn
16006     \or:
16007       \_fp_case_return_o:Nw \c_nan_fp
16008     \else:
16009       \exp_after:wN \_fp_pow_neg:www
16010       \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
16011     \fi:
16012     {
16013       \if_meaning:w 1 #1
16014         \exp_after:wN \_fp_pow_normal_o:ww
16015       \else:
16016         \exp_after:wN \_fp_pow_zero_or_inf:ww
16017       \fi:
16018       \s__fp \_fp_chk:w #1#2#3;
16019     }

```

```

16020     { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
16021     \s__fp \__fp_chk:w #4#5#6;
16022 }

```

(End definition for __fp_~o:ww.)

__fp_pow_zero_or_inf:ww Raising -0 or $-\infty$ to `nan` yields `nan`. For other powers, the result is $+0$ if 0 is raised to a positive power or ∞ to a negative power, and $+\infty$ otherwise. Thus, if the type of a and the sign of b coincide, the result is 0 , since those conveniently take the same possible values, 0 and 2 . Otherwise, either $a = \pm\infty$ and $b > 0$ and the result is $+\infty$, or $a = \pm 0$ with $b < 0$ and we have a division by zero unless $b = -\infty$.

```

16023 \cs_new:Npn \__fp_pow_zero_or_inf:ww
16024   \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
16025   {
16026     \if_meaning:w 1 #4
16027       \__fp_case_return_same_o:w
16028     \fi:
16029     \if_meaning:w #1 #4
16030       \__fp_case_return_o:Nw \c_zero_fp
16031     \fi:
16032     \if_meaning:w 2 #1
16033       \__fp_case_return_o:Nw \c_inf_fp
16034     \fi:
16035     \if_meaning:w 2 #3
16036       \__fp_case_return_o:Nw \c_inf_fp
16037     \else:
16038       \__fp_case_use:nw
16039       {
16040         \__fp_division_by_zero_o:NNww \c_inf_fp ^
16041         \s__fp \__fp_chk:w #1 #2 ;
16042       }
16043     \fi:
16044     \s__fp \__fp_chk:w #3#4
16045   }

```

(End definition for __fp_pow_zero_or_inf:ww.)

__fp_pow_normal_o:ww We have in front of us a , and $b \neq 0$, we know that a is a normal number, and we wish to compute $|a|^b$. If $|a| = 1$, we return 1 , unless $a = -1$ and b is `nan`. Indeed, returning 1 at this point would wrongly raise “invalid” when the sign is considered. If $|a| \neq 1$, test the type of b :

- 0 Impossible, we already filtered $b = \pm 0$.
- 1 Call `__fp_pow_npos_o:Nww`.
- 2 Return $+\infty$ or $+0$ depending on the sign of b and whether the exponent of a is positive or not.
- 3 Return b .

```

16046 \cs_new:Npn \__fp_pow_normal_o:ww
16047   \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
16048   {
16049     \if_int_compare:w \__fp_str_if_eq_x:nn { #2 #3 }

```

```

16050         { 1 {1000} {0000} {0000} {0000} } = 0 \exp_stop_f:
16051     \if_int_compare:w #4 #1 = 32 \exp_stop_f:
16052         \exp_after:wN \__fp_case_return_ii_o:ww
16053     \fi:
16054     \__fp_case_return_o:Nww \c_one_fp
16055 \fi:
16056 \if_case:w #4 \exp_stop_f:
16057 \or:
16058     \exp_after:wN \__fp_pow_npos_o:Nww
16059     \exp_after:wN #5
16060 \or:
16061     \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
16062     \if_int_compare:w #2 > 0 \exp_stop_f:
16063         \exp_after:wN \__fp_case_return_o:Nww
16064         \exp_after:wN \c_inf_fp
16065     \else:
16066         \exp_after:wN \__fp_case_return_o:Nww
16067         \exp_after:wN \c_zero_fp
16068     \fi:
16069 \or:
16070     \__fp_case_return_ii_o:ww
16071 \fi:
16072 \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
16073 \s__fp \__fp_chk:w #4 #5
16074 }

```

(End definition for __fp_pow_normal_o:ww.)

__fp_pow_npos_o:Nww

We now know that $a \neq \pm 1$ is a normal number, and b is a normal number too. We want to compute $|a|^b = (|x| \cdot 10^n)^{y \cdot 10^p} = \exp((\ln|x| + n \ln(10)) \cdot y \cdot 10^p) = \exp(z)$. To compute the exponential accurately, we need to know the digits of z up to the 16-th position. Since the exponential of 10^5 is infinite, we only need at most 21 digits, hence the fixed point result of __fp_ln_o:w is precise enough for our needs. Start an integer expression for the decimal exponent of $e^{|z|}$. If z is negative, negate that decimal exponent, and prepare to take the inverse when converting from the fixed point to the floating point result.

```

16075 \cs_new:Npn \__fp_pow_npos_o:Nww #1 \s__fp \__fp_chk:w 1#2#3
16076 {
16077     \exp_after:wN \__fp_sanitize:Nw
16078     \exp_after:wN 0
16079     \int_value:w
16080     \if:w #1 \if_int_compare:w #3 > 0 \exp_stop_f: 0 \else: 2 \fi:
16081     \exp_after:wN \__fp_pow_npos_aux:NNnw
16082     \exp_after:wN +
16083     \exp_after:wN \__fp_fixed_to_float_o:wN
16084     \else:
16085     \exp_after:wN \__fp_pow_npos_aux:NNnw
16086     \exp_after:wN -
16087     \exp_after:wN \__fp_fixed_inv_to_float_o:wN
16088     \fi:
16089     {#3}
16090 }

```

(End definition for __fp_pow_npos_o:Nww.)

__fp_pow_npos_aux:NNnww The first argument is the conversion function from fixed point to float. Then comes an exponent and the 4 brace groups of x , followed by b . Compute $-\ln(x)$.

```

16091 \cs_new:Npn \__fp_pow_npos_aux:NNnww #1#2#3#4#5; \s_fp \__fp_chk:w 1#6#7#8;
16092 {
16093   #1
16094   \__fp_int_eval:w
16095   \__fp_ln_significand:NNNNnnnN #4#5
16096   \__fp_pow_exponent:wnN {#3}
16097   \__fp_fixed_mul:wwN #8 {0000}{0000} ;
16098   \__fp_pow_B:wwN #7;
16099   #1 #2 0 % fixed_to_float_o:wn
16100 }
16101 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
16102 {
16103   \if_int_compare:w #2 > 0 \exp_stop_f:
16104   \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
16105   \exp_after:wN +
16106   \else:
16107   \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % -(\ln|\ln(10) + (-\ln(x)))
16108   \exp_after:wN -
16109   \fi:
16110   #2; #1;
16111 }
16112 \cs_new:Npn \__fp_pow_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
16113 { %^A todo: use that in ln.
16114   \exp_after:wN \__fp_fixed_mul_after:wwN
16115   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
16116   \exp_after:wN \__fp_pack:NNNNNw
16117   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
16118   #1#2*23025 - #1 #3
16119   \exp_after:wN \__fp_pack:NNNNNw
16120   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
16121   #1 #2*8509 - #1 #4
16122   \exp_after:wN \__fp_pack:NNNNNw
16123   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
16124   #1 #2*2994 - #1 #5
16125   \exp_after:wN \__fp_pack:NNNNNw
16126   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
16127   #1 #2*0456 - #1 #6
16128   \exp_after:wN \__fp_pack:NNNNNw
16129   \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
16130   #1 #2*8401 - #1 #7
16131   #1 ( #2*7991 - #8 ) / 1 0000 ; ;
16132 }
16133 \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
16134 {
16135   \if_int_compare:w #7 < 0 \exp_stop_f:
16136   \exp_after:wN \__fp_pow_C_neg:w \int_value:w -
16137   \else:
16138   \if_int_compare:w #7 < 22 \exp_stop_f:
16139   \exp_after:wN \__fp_pow_C_pos:w \int_value:w
16140   \else:
16141   \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
16142   \fi:

```

```

16143 \fi:
16144 #7 \exp_after:wN ;
16145 \int_value:w \__fp_int_eval:w 10 0000 + #1 \__fp_int_eval_end:
16146 #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^A todo: how many 0?
16147 }
16148 \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
16149 {
16150 + 2 * \c__fp_max_exponent_int
16151 \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl
16152 }
16153 \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
16154 {
16155 \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
16156 \prg_replicate:nn {#1} {0}
16157 }
16158 \cs_new:Npn \__fp_pow_C_pos:w #1; 1
16159 { \__fp_pow_C_pos_loop:wN #1; }
16160 \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
16161 {
16162 \if_meaning:w 0 #1
16163 \exp_after:wN \__fp_pow_C_pack:w
16164 \exp_after:wN #2
16165 \else:
16166 \if_meaning:w 0 #2
16167 \exp_after:wN \__fp_pow_C_pos_loop:wN \int_value:w
16168 \else:
16169 \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
16170 \fi:
16171 \__fp_int_eval:w #1 - 1 \exp_after:wN ;
16172 \fi:
16173 }
16174 \cs_new:Npn \__fp_pow_C_pack:w
16175 { \exp_after:wN \__fp_exp_large_v:wN \c__fp_one_fixed_tl }

```

(End definition for __fp_pow_npos_aux:NNnww.)

__fp_pow_neg:www
__fp_pow_neg_aux:wNN

This function is followed by three floating point numbers: a^b , $a \in [-\infty, -0]$, and b . If b is an even integer (case -1), $a^b = a^b$. If b is an odd integer (case 0), $a^b = -a^b$, obtained by a call to __fp_pow_neg_aux:wNN. Otherwise, the sign is undefined. This is invalid, unless a^b turns out to be $+0$ or nan , in which case we return that as a^b . In particular, since the underflow detection occurs before __fp_pow_neg:www is called, $(-0.1)**(12345.67)$ gives $+0$ rather than complaining that the sign is not defined.

```

16176 \cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
16177 {
16178 \if_case:w \__fp_pow_neg_case:w #4 ;
16179 \exp_after:wN \__fp_pow_neg_aux:wNN
16180 \or:
16181 \if_int_compare:w \__fp_int_eval:w #1 / 2 = 1 \exp_stop_f:
16182 \__fp_invalid_operation_o:Nww ^ #3; #4;
16183 \exp:w \exp_end_continue_f:w
16184 \exp_after:wN \exp_after:wN
16185 \exp_after:wN \__fp_use_none_until_s:w
16186 \fi:
16187 \fi:

```



```

16188     \__fp_exp_after_o:w
16189     \s__fp \__fp_chk:w #1#2;
16190 }
16191 \cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
16192 {
16193     \exp_after:wN \__fp_exp_after_o:w
16194     \exp_after:wN \s__fp
16195     \exp_after:wN \__fp_chk:w
16196     \exp_after:wN #2
16197     \int_value:w \__fp_int_eval:w 2 - #3 \__fp_int_eval_end:
16198 }

```

(End definition for __fp_pow_neg:www and __fp_pow_neg_aux:wNN.)

```

\__fp_pow_neg_case:w
\__fp_pow_neg_case_aux:nnnnn
\__fp_pow_neg_case_aux:Nnnw

```

This function expects a floating point number, and determines its “parity”. It should be used after \if_case:w or in an integer expression. It gives -1 if the number is an even integer, 0 if the number is an odd integer, and 1 otherwise. Zeros and $\pm\infty$ are even (because very large finite floating points are even), while `nan` is a non-integer. The sign of normal numbers is irrelevant to parity. After __fp_decimate:nNnnnn the argument `#1` of __fp_pow_neg_case_aux:Nnnw is a rounding digit, 0 if and only if the number was an integer, and `#3` is the 8 least significant digits of that integer.

```

16199 \cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
16200 {
16201     \if_case:w #1 \exp_stop_f:
16202         -1
16203     \or: \__fp_pow_neg_case_aux:nnnnn #3
16204     \or: -1
16205     \else: 1
16206     \fi:
16207     \exp_stop_f:
16208 }
16209 \cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
16210 {
16211     \if_int_compare:w #1 > \c__fp_prec_int
16212         -1
16213     \else:
16214         \__fp_decimate:nNnnnn { \c__fp_prec_int - #1 }
16215         \__fp_pow_neg_case_aux:Nnnw
16216         {#2} {#3} {#4} {#5}
16217     \fi:
16218 }
16219 \cs_new:Npn \__fp_pow_neg_case_aux:Nnnw #1#2#3#4 ;
16220 {
16221     \if_meaning:w 0 #1
16222         \if_int_odd:w #3 \exp_stop_f:
16223         0
16224     \else:
16225         -1
16226     \fi:
16227     \else:
16228         1
16229     \fi:
16230 }

```

(End definition for `__fp_pow_neg_case:w`, `__fp_pow_neg_case_aux:nnnnn`, and `__fp_pow_neg_case_aux:Nnnw`.)

16231 \langle /initex | package \rangle

31 l3fp-trig Implementation

16232 \langle *initex | package \rangle

16233 \langle @@=fp \rangle

```

\__fp_parse_word_acos:N
\__fp_parse_word_acosd:N
\__fp_parse_word_acsc:N
\__fp_parse_word_acscd:N
\__fp_parse_word_asec:N
\__fp_parse_word_asecd:N
\__fp_parse_word_asin:N
\__fp_parse_word_asind:N
\__fp_parse_word_cos:N
\__fp_parse_word_cosd:N
\__fp_parse_word_cot:N
\__fp_parse_word_cotd:N
\__fp_parse_word_csc:N
\__fp_parse_word_cscd:N
\__fp_parse_word_sec:N
\__fp_parse_word_secd:N
\__fp_parse_word_sin:N
\__fp_parse_word_sind:N
\__fp_parse_word_tan:N
\__fp_parse_word_tand:N

```

Unary functions.

```

16234 \tl_map_inline:nn
16235 {
16236   {acos} {acsc} {asec} {asin}
16237   {cos} {cot} {csc} {sec} {sin} {tan}
16238 }
16239 {
16240   \cs_new:cpx { __fp_parse_word_#1:N }
16241   {
16242     \exp_not:N \__fp_parse_unary_function:NNN
16243     \exp_not:c { __fp_#1_o:w }
16244     \exp_not:N \use_i:nn
16245   }
16246   \cs_new:cpx { __fp_parse_word_#1d:N }
16247   {
16248     \exp_not:N \__fp_parse_unary_function:NNN
16249     \exp_not:c { __fp_#1_o:w }
16250     \exp_not:N \use_ii:nn
16251   }
16252 }

```

(End definition for `__fp_parse_word_acos:N` and others.)

```

\__fp_parse_word_acot:N
\__fp_parse_word_acotd:N
\__fp_parse_word_atan:N
\__fp_parse_word_atand:N

```

Those functions may receive a variable number of arguments.

```

16253 \cs_new:Npn \__fp_parse_word_acot:N
16254 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
16255 \cs_new:Npn \__fp_parse_word_acotd:N
16256 { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
16257 \cs_new:Npn \__fp_parse_word_atan:N
16258 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
16259 \cs_new:Npn \__fp_parse_word_atand:N
16260 { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }

```

(End definition for `__fp_parse_word_acot:N` and others.)

31.1 Direct trigonometric functions

The approach for all trigonometric functions (sine, cosine, tangent, cotangent, cosecant, and secant), with arguments given in radians or in degrees, is the same.

- Filter out special cases (± 0 , $\pm \infty$ and NaN).
- Keep the sign for later, and work with the absolute value $|x|$ of the argument.

- Small numbers ($|x| < 1$ in radians, $|x| < 10$ in degrees) are converted to fixed point numbers (and to radians if $|x|$ is in degrees).
- For larger numbers, we need argument reduction. Subtract a multiple of $\pi/2$ (in degrees, 90) to bring the number to the range to $[0, \pi/2)$ (in degrees, $[0, 90)$).
- Reduce further to $[0, \pi/4]$ (in degrees, $[0, 45]$) using $\sin x = \cos(\pi/2 - x)$, and when working in degrees, convert to radians.
- Use the appropriate power series depending on the octant $\lfloor \frac{x}{\pi/4} \rfloor \bmod 8$ (in degrees, the same formula with $\pi/4 \rightarrow 45$), the sign, and the function to compute.

31.1.1 Filtering special cases

`__fp_sin_o:w` This function, and its analogs for `cos`, `csc`, `sec`, `tan`, and `cot` instead of `sin`, are followed either by `\use_i:nn` and a float in radians or by `\use_ii:nn` and a float in degrees. The sine of ± 0 or NaN is the same float. The sine of $\pm\infty$ raises an invalid operation exception with the appropriate function name. Otherwise, call the `trig` function to perform argument reduction and if necessary convert the reduced argument to radians. Then, `__fp_sin_series_o:NNwww` is called to compute the Taylor series: this function receives a sign #3, an initial octant of 0, and the function `__fp_ep_to_float_o:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since $\sin(x) = \#3 \sin|x|$.

```

16261 \cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
16262 {
16263   \if_case:w #2 \exp_stop_f:
16264     \__fp_case_return_same_o:w
16265   \or: \__fp_case_use:nw
16266     {
16267       \__fp_trig:NNNNwn #1 \__fp_sin_series_o:NNwww
16268       \__fp_ep_to_float_o:wwN #3 0
16269     }
16270   \or: \__fp_case_use:nw
16271     { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
16272   \else: \__fp_case_return_same_o:w
16273   \fi:
16274   \s__fp \__fp_chk:w #2 #3 #4;
16275 }
```

(End definition for `__fp_sin_o:w`.)

`__fp_cos_o:w` The cosine of ± 0 is 1. The cosine of $\pm\infty$ raises an invalid operation exception. The cosine of NaN is itself. Otherwise, the `trig` function reduces the argument to at most half a right-angle and converts if necessary to radians. We then call the same series as for sine, but using a positive sign 0 regardless of the sign of x , and with an initial octant of 2, because $\cos(x) = +\sin(\pi/2 + |x|)$.

```

16276 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
16277 {
16278   \if_case:w #2 \exp_stop_f:
16279     \__fp_case_return_o:Nw \c_one_fp
16280   \or: \__fp_case_use:nw
16281     {
16282       \__fp_trig:NNNNwn #1 \__fp_sin_series_o:NNwww
```

```

16283         \__fp_ep_to_float_o:wwN 0 2
16284     }
16285     \or:   \__fp_case_use:nw
16286           { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
16287     \else: \__fp_case_return_same_o:w
16288     \fi:
16289     \s__fp \__fp_chk:w #2 #3;
16290 }

```

(End definition for __fp_cos_o:w.)

__fp_csc_o:w The cosecant of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see __fp_cot_zero_o:Nfw defined below), which requires the function name. The cosecant of $\pm\infty$ raises an invalid operation exception. The cosecant of NaN is itself. Otherwise, the `trig` function performs the argument reduction, and converts if necessary to radians before calling the same series as for sine, using the sign #3, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because $\csc(x) = \#3(\sin|x|)^{-1}$.

```

16291 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
16292 {
16293     \if_case:w #2 \exp_stop_f:
16294         \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
16295     \or:   \__fp_case_use:nw
16296           {
16297             \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
16298             \__fp_ep_inv_to_float_o:wwN #3 0
16299           }
16300     \or:   \__fp_case_use:nw
16301           { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
16302     \else: \__fp_case_return_same_o:w
16303     \fi:
16304     \s__fp \__fp_chk:w #2 #3 #4;
16305 }

```

(End definition for __fp_csc_o:w.)

__fp_sec_o:w The secant of ± 0 is 1. The secant of $\pm\infty$ raises an invalid operation exception. The secant of NaN is itself. Otherwise, the `trig` function reduces the argument and turns it to radians before calling the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```

16306 \cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
16307 {
16308     \if_case:w #2 \exp_stop_f:
16309         \__fp_case_return_o:Nw \c_one_fp
16310     \or:   \__fp_case_use:nw
16311           {
16312             \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwww
16313             \__fp_ep_inv_to_float_o:wwN 0 2
16314           }
16315     \or:   \__fp_case_use:nw
16316           { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
16317     \else: \__fp_case_return_same_o:w
16318     \fi:
16319     \s__fp \__fp_chk:w #2 #3;

```

```
16320 }
```

(End definition for `_fp_sec_o:w`.)

`_fp_tan_o:w` The tangent of ± 0 or NaN is the same floating point number. The tangent of $\pm\infty$ raises an invalid operation exception. Once more, the `trig` function does the argument reduction step and conversion to radians before calling `_fp_tan_series_o:NNwww`, with a sign #3 and an initial octant of 1 (this shift is somewhat arbitrary). See `_fp_cot_o:w` for an explanation of the 0 argument.

```
16321 \cs_new:Npn \_fp_tan_o:w #1 \s__fp \_fp_chk:w #2#3#4; @
16322 {
16323   \if_case:w #2 \exp_stop_f:
16324     \_fp_case_return_same_o:w
16325   \or: \_fp_case_use:nw
16326     {
16327       \_fp_trig:NNNNwn #1
16328       \_fp_tan_series_o:NNwww 0 #3 1
16329     }
16330   \or: \_fp_case_use:nw
16331     { \_fp_invalid_operation_o:fw { #1 { tan } { tand } } }
16332   \else: \_fp_case_return_same_o:w
16333   \fi:
16334   \s__fp \_fp_chk:w #2 #3 #4;
16335 }
```

(End definition for `_fp_tan_o:w`.)

`_fp_cot_o:w` The cotangent of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `_fp_cot_zero_o:Nfw`). The cotangent of $\pm\infty$ raises an invalid operation exception. The cotangent of NaN is itself. We use $\cot x = -\tan(\pi/2 + x)$, and the initial octant for the tangent was chosen to be 1, so the octant here starts at 3. The change in sign is obtained by feeding `_fp_tan_series_o:NNwww` two signs rather than just the sign of the argument: the first of those indicates whether we compute tangent or cotangent. Those signs are eventually combined.

```
16336 \cs_new:Npn \_fp_cot_o:w #1 \s__fp \_fp_chk:w #2#3#4; @
16337 {
16338   \if_case:w #2 \exp_stop_f:
16339     \_fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
16340   \or: \_fp_case_use:nw
16341     {
16342       \_fp_trig:NNNNwn #1
16343       \_fp_tan_series_o:NNwww 2 #3 3
16344     }
16345   \or: \_fp_case_use:nw
16346     { \_fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
16347   \else: \_fp_case_return_same_o:w
16348   \fi:
16349   \s__fp \_fp_chk:w #2 #3 #4;
16350 }
16351 \cs_new:Npn \_fp_cot_zero_o:Nfw #1#2#3 \fi:
16352 {
16353   \fi:
16354   \token_if_eq_meaning:NNTF 0 #1
16355     { \exp_args:NNf \_fp_division_by_zero_o:Nnw \c_inf_fp }
```

```

16356      { \exp_args:Nnf \_fp_division_by_zero_o:Nnw \c_minus_inf_fp }
16357      {#2}
16358    }

```

(End definition for `_fp_cot_o:w` and `_fp_cot_zero_o:Nfw`.)

31.1.2 Distinguishing small and large arguments

`_fp_trig:NNNNNwn` The first argument is `\use_i:nn` if the operand is in radians and `\use_ii:nn` if it is in degrees. Arguments #2 to #5 control what trigonometric function we compute, and #6 to #8 are pieces of a normal floating point number. Call the `_series` function #2, with arguments #3, either a conversion function (`_fp_ep_to_float_o:wN` or `_fp_ep_inv_to_float_o:wN`) or a sign 0 or 2 when computing tangent or cotangent; #4, a sign 0 or 2; the octant, computed in an integer expression starting with #5 and stopped by a period; and a fixed point number obtained from the floating point number by argument reduction (if necessary) and conversion to radians (if necessary). Any argument reduction adjusts the octant accordingly by leaving a (positive) shift into its integer expression. Let us explain the integer comparison. Two of the four `\exp_after:wN` are expanded, the expansion hits the test, which is true if the float is at least 1 when working in radians, and at least 10 when working in degrees. Then one of the remaining `\exp_after:wN` hits #1, which picks the `trig` or `trigd` function in whichever branch of the conditional was taken. The final `\exp_after:wN` closes the conditional. At the end of the day, a number is **large** if it is ≥ 1 in radians or ≥ 10 in degrees, and **small** otherwise. All four `trig/trigd` auxiliaries receive the operand as an extended-precision number.

```

16359 \cs_new:Npn \_fp_trig:NNNNNwn #1#2#3#4#5 \s_fp \_fp_chk:w 1#6#7#8;
16360 {
16361   \exp_after:wN #2
16362   \exp_after:wN #3
16363   \exp_after:wN #4
16364   \int_value:w \_fp_int_eval:w #5
16365   \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
16366   \if_int_compare:w #7 > #1 0 1 \exp_stop_f:
16367     #1 \_fp_trig_large:ww \_fp_trigd_large:ww
16368   \else:
16369     #1 \_fp_trig_small:ww \_fp_trigd_small:ww
16370   \fi:
16371   #7,#8{0000}{0000};
16372 }

```

(End definition for `_fp_trig:NNNNNwn`.)

31.1.3 Small arguments

`_fp_trig_small:ww` This receives a small extended-precision number in radians and converts it to a fixed point number. Some trailing digits may be lost in the conversion, so we keep the original floating point number around: when computing sine or tangent (or their inverses), the last step is to multiply by the floating point number (as an extended-precision number) rather than the fixed point number. The period serves to end the integer expression for the octant.

```

16373 \cs_new:Npn \_fp_trig_small:ww #1,#2;
16374 { \_fp_ep_to_fixed:wwn #1,#2; . #1,#2; }

```

(End definition for `_fp_trig_small:ww`.)

`__fp_trigd_small:ww` Convert the extended-precision number to radians, then call `__fp_trig_small:ww` to massage it in the form appropriate for the `_series` auxiliary.

```

16375 \cs_new:Npn \__fp_trigd_small:ww #1,#2;
16376 {
16377   \__fp_ep_mul_raw:wwwN
16378   -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
16379   \__fp_trig_small:ww
16380 }

```

(End definition for `__fp_trigd_small:ww`.)

31.1.4 Argument reduction in degrees

`__fp_trigd_large:ww` Note that $25 \times 360 = 9000$, so $10^{k+1} \equiv 10^k \pmod{360}$ for $k \geq 3$. When the exponent `#1` is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is $100 \times \langle block_1 \rangle \cdots \langle block_3 \rangle . \langle block_4 \rangle \cdots \langle block_7 \rangle$, or is equal to it modulo 360 if the exponent `#1` is very large. The first auxiliary finds $\langle block_1 \rangle + \langle block_2 \rangle \pmod{9}$, a single digit, and prepends it to the 4 digits of $\langle block_3 \rangle$. It also unpacks $\langle block_4 \rangle$ and grabs the 4 digits of $\langle block_7 \rangle$. The second auxiliary grabs the $\langle block_3 \rangle$ plus any contribution from the first two blocks as `#1`, the first digit of $\langle block_4 \rangle$ (just after the decimal point in hundreds of degrees) as `#2`, and the three other digits as `#3`. It finds the quotient and remainder of `#1#2` modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before `#3` to form a new $\langle block_4 \rangle$. The resulting fixed point number is $x \in [0, 0.9]$. If $x \geq 0.45$, we add 1 to the octant and feed $0.9 - x$ with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to `__fp_trigd_small:ww`. Otherwise, we feed it x with an exponent of 2. The third auxiliary also discards digits which were not packed into the various $\langle blocks \rangle$. Since the original exponent `#1` is at least 2, those are all 0 and no precision is lost (`#6` and `#7` are four 0 each).

```

16381 \cs_new:Npn \__fp_trigd_large:ww #1, #2#3#4#5#6#7;
16382 {
16383   \exp_after:wN \__fp_pack_eight:wNNNNNNNN
16384   \exp_after:wN \__fp_pack_eight:wNNNNNNNN
16385   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
16386   \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
16387   \exp_after:wN \__fp_trigd_large_auxi:nnnnwNNNN
16388   \exp_after:wN ;
16389   \exp:w \exp_end_continue_f:w
16390   \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
16391   #2#3#4#5#6#7 0000 0000 0000 !
16392 }
16393 \cs_new:Npn \__fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
16394 {
16395   \exp_after:wN \__fp_trigd_large_auxii:wNw
16396   \int_value:w \__fp_int_eval:w #1 + #2
16397   - (#1 + #2 - 4) / 9 * 9 \__fp_int_eval_end:
16398   #3;
16399   #4; #5{#6#7#8#9};
16400 }
16401 \cs_new:Npn \__fp_trigd_large_auxii:wNw #1; #2#3;
16402 {

```

```

16403     + (#1#2 - 4) / 9 * 2
16404     \exp_after:wN \__fp_trigd_large_auxiii:www
16405     \int_value:w \__fp_int_eval:w #1#2
16406     - (#1#2 - 4) / 9 * 9 \__fp_int_eval_end: #3 ;
16407 }
16408 \cs_new:Npn \__fp_trigd_large_auxiii:www #1; #2; #3!
16409 {
16410     \if_int_compare:w #1 < 4500 \exp_stop_f:
16411     \exp_after:wN \__fp_use_i_until_s:nw
16412     \exp_after:wN \__fp_fixed_continue:wn
16413     \else:
16414     + 1
16415     \fi:
16416     \__fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
16417     {#1#2}{0000}{0000};
16418     { \__fp_trigd_small:ww 2, }
16419 }

```

(End definition for `__fp_trigd_large:ww` and others.)

31.1.5 Argument reduction in radians

Arguments greater or equal to 1 need to be reduced to a range where we only need a few terms of the Taylor series. We reduce to the range $[0, 2\pi]$ by subtracting multiples of 2π , then to the smaller range $[0, \pi/2]$ by subtracting multiples of $\pi/2$ (keeping track of how many times $\pi/2$ is subtracted), then to $[0, \pi/4]$ by mapping $x \rightarrow \pi/2 - x$ if appropriate. When the argument is very large, say, 10^{100} , an equally large multiple of 2π must be subtracted, hence we must work with a very good approximation of 2π in order to get a sensible remainder modulo 2π .

Specifically, we multiply the argument by an approximation of $1/(2\pi)$ with 10048 digits, then discard the integer part of the result, keeping 52 digits of the fractional part. From the fractional part of $x/(2\pi)$ we deduce the octant (quotient of the first three digits by 125). We then multiply by 8 or -8 (the latter when the octant is odd), ignore any integer part (related to the octant), and convert the fractional part to an extended precision number, before multiplying by $\pi/4$ to convert back to a value in radians in $[0, \pi/4]$.

It is possible to prove that given the precision of floating points and their range of exponents, the 52 digits may start at most with 24 zeros. The 5 last digits are affected by carries from computations which are not done, hence we are left with at least $52 - 24 - 5 = 23$ significant digits, enough to round correctly up to $0.6 \cdot \text{ulp}$ in all cases.

`__fp_trig_inverse_two_pi:` This macro expands to `, , !` or `, !` followed by 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we later need, 52, plus 12 (4 – 1 groups of 4 digits). We store the decimals as a control sequence name, and convert it to a token list when required: strings take up less memory than their token list representation.

```

16420 \cs_new:Npx \__fp_trig_inverse_two_pi:
16421 {
16422     \exp_not:n { \exp_after:wN \use_none:n \token_to_str:N }
16423     \cs:w , , !
16424     00000000000000000000159154943091895335768883763372514362034459645740 ~
16425     4564487476673440588967976342265350901138027662530859560728427267 ~

```


16426 5795803689291184611457865287796741073169983922923996693740907757 ~
16427 3077746396925307688717392896217397661693362390241723629011832380 ~
16428 1142226997557159404618900869026739561204894109369378440855287230 ~
16429 9994644340024867234773945961089832309678307490616698646280469944 ~
16430 8652187881574786566964241038995874139348609983868099199962442875 ~
16431 5851711788584311175187671605465475369880097394603647593337680593 ~
16432 0249449663530532715677550322032477781639716602294674811959816584 ~
16433 0606016803035998133911987498832786654435279755070016240677564388 ~
16434 8495713108801221993761476813777647378906330680464579784817613124 ~
16435 2731406996077502450029775985708905690279678513152521001631774602 ~
16436 0924811606240561456203146484089248459191435211575407556200871526 ~
16437 6068022171591407574745827225977462853998751553293908139817724093 ~
16438 5825479707332871904069997590765770784934703935898280871734256403 ~
16439 6689511662545705943327631268650026122717971153211259950438667945 ~
16440 0376255608363171169525975812822494162333431451061235368785631136 ~
16441 3669216714206974696012925057833605311960859450983955671870995474 ~
16442 6510431623815517580839442979970999505254387566129445883306846050 ~
16443 7852915151410404892988506388160776196993073410389995786918905980 ~
16444 9373777206187543222718930136625526123878038753888110681406765434 ~
16445 0828278526933426799556070790386060352738996245125995749276297023 ~
16446 5940955843011648296411855777124057544494570217897697924094903272 ~
16447 9477021664960356531815354400384068987471769158876319096650696440 ~
16448 4776970687683656778104779795450353395758301881838687937766124814 ~
16449 9530599655802190835987510351271290432315804987196868777594656634 ~
16450 6221034204440855497850379273869429353661937782928735937843470323 ~
16451 0237145837923557118636341929460183182291964165008783079331353497 ~
16452 7909974586492902674506098936890945883050337030538054731232158094 ~
16453 3197676032283131418980974982243833517435698984750103950068388003 ~
16454 9786723599608024002739010874954854787923568261139948903268997427 ~
16455 0834961149208289037767847430355045684560836714793084567233270354 ~
16456 8539255620208683932409956221175331839402097079357077496549880868 ~
16457 6066360968661967037474542102831219251846224834991161149566556037 ~
16458 9696761399312829960776082779901007830360023382729879085402387615 ~
16459 5744543092601191005433799838904654921248295160707285300522721023 ~
16460 6017523313173179759311050328155109373913639645305792607180083617 ~
16461 9548767246459804739772924481092009371257869183328958862839904358 ~
16462 6866663975673445140950363732719174311388066383072592302759734506 ~
16463 054821278037065337783032170987734966568490800326988506741791464 ~
16464 6835082816168533143361607309951498531198197337584442098416559541 ~
16465 5225064339431286444038388356150879771645017064706751877456059160 ~
16466 8716857857939226234756331711132998655941596890719850688744230057 ~
16467 5191977056900382183925622033874235362568083541565172971088117217 ~
16468 9593683256488518749974870855311659830610139214454460161488452770 ~
16469 2511411070248521739745103866736403872860099674893173561812071174 ~
16470 0478899368886556923078485023057057144063638632023685201074100574 ~
16471 8592281115721968003978247595300166958522123034641877365043546764 ~
16472 6456565971901123084767099309708591283646669191776938791433315566 ~
16473 5066981321641521008957117286238426070678451760111345080069947684 ~
16474 2235698962488051577598095339708085475059753626564903439445420581 ~
16475 7886435683042000315095594743439252544850674914290864751442303321 ~
16476 3324569511634945677539394240360905438335528292434220349484366151 ~
16477 4663228602477666660495314065734357553014090827988091478669343492 ~
16478 2737602634997829957018161964321233140475762897484082891174097478 ~
16479 2637899181699939487497715198981872666294601830539583275209236350 ~

16480 6853889228468247259972528300766856937583659722919824429747406163 ~
16481 8183113958306744348516928597383237392662402434501997809940402189 ~
16482 6134834273613676449913827154166063424829363741850612261086132119 ~
16483 9863346284709941839942742955915628333990480382117501161211667205 ~
16484 1912579303552929241134403116134112495318385926958490443846807849 ~
16485 0973982808855297045153053991400988698840883654836652224668624087 ~
16486 2540140400911787421220452307533473972538149403884190586842311594 ~
16487 6322744339066125162393106283195323883392131534556381511752035108 ~
16488 7459558201123754359768155340187407394340363397803881721004531691 ~
16489 8295194879591767395417787924352761740724605939160273228287946819 ~
16490 3649128949714953432552723591659298072479985806126900733218844526 ~
16491 7943350455801952492566306204876616134365339920287545208555344144 ~
16492 0990512982727454659118132223284051166615650709837557433729548631 ~
16493 2041121716380915606161165732000083306114606181280326258695951602 ~
16494 4632166138576614804719932707771316441201594960110632830520759583 ~
16495 4850305079095584982982186740289838551383239570208076397550429225 ~
16496 9847647071016426974384504309165864528360324933604354657237557916 ~
16497 1366324120457809969715663402215880545794313282780055246132088901 ~
16498 8742121092448910410052154968097113720754005710963406643135745439 ~
16499 9159769435788920793425617783022237011486424925239248728713132021 ~
16500 7667360756645598272609574156602343787436291321097485897150713073 ~
16501 9104072643541417970572226547980381512759579124002534468048220261 ~
16502 7342299001020483062463033796474678190501811830375153802879523433 ~
16503 4195502135689770912905614317878792086205744999257897569018492103 ~
16504 2420647138519113881475640209760554895793785141404145305151583964 ~
16505 2823265406020603311891586570272086250269916393751527887360608114 ~
16506 5569484210322407772727421651364234366992716340309405307480652685 ~
16507 0930165892136921414312937134106157153714062039784761842650297807 ~
16508 8606266969960809184223476335047746719017450451446166382846208240 ~
16509 8673595102371302904443779408535034454426334130626307459513830310 ~
16510 2293146934466832851766328241515210179422644395718121717021756492 ~
16511 1964449396532222187658488244511909401340504432139858628621083179 ~
16512 3939608443898019147873897723310286310131486955212620518278063494 ~
16513 5711866277825659883100535155231665984394090221806314454521212978 ~
16514 9734471488741258268223860236027109981191520568823472398358013366 ~
16515 0683786328867928619732367253606685216856320119489780733958419190 ~
16516 6659583867852941241871821727987506103946064819585745620060892122 ~
16517 8416394373846549589932028481236433466119707324309545859073361878 ~
16518 6290631850165106267576851216357588696307451999220010776676830946 ~
16519 9814975622682434793671310841210219520899481912444048751171059184 ~
16520 4139907889455775184621619041530934543802808938628073237578615267 ~
16521 7971143323241969857805637630180884386640607175368321362629671224 ~
16522 2609428540110963218262765120117022552929289655594608204938409069 ~
16523 0760692003954646191640021567336017909631872891998634341086903200 ~
16524 5796637103128612356988817640364252540837098108148351903121318624 ~
16525 7228181050845123690190646632235938872454630737272808789830041018 ~
16526 9485913673742589418124056729191238003306344998219631580386381054 ~
16527 2457893450084553280313511884341007373060595654437362488771292628 ~
16528 9807423539074061786905784443105274262641767830058221486462289361 ~
16529 9296692992033046693328438158053564864073184440599549689353773183 ~
16530 6726613130108623588021288043289344562140479789454233736058506327 ~
16531 0439981932635916687341943656783901281912202816229500333012236091 ~
16532 8587559201959081224153679499095448881099758919890811581163538891 ~
16533 6339402923722049848375224236209100834097566791710084167957022331 ~

```

16534 7897107102928884897013099533995424415335060625843921452433864640 ~
16535 3432440657317477553405404481006177612569084746461432976543900008 ~
16536 3826521145210162366431119798731902751191441213616962045693602633 ~
16537 6102355962140467029012156796418735746835873172331004745963339773 ~
16538 2477044918885134415363760091537564267438450166221393719306748706 ~
16539 2881595464819775192207710236743289062690709117919412776212245117 ~
16540 2354677115640433357720616661564674474627305622913332030953340551 ~
16541 3841718194605321501426328000879551813296754972846701883657425342 ~
16542 5016994231069156343106626043412205213831587971115075454063290657 ~
16543 0248488648697402872037259869281149360627403842332874942332178578 ~
16544 7750735571857043787379693402336902911446961448649769719434527467 ~
16545 4429603089437192540526658890710662062575509930379976658367936112 ~
16546 8137451104971506153783743579555867972129358764463093757203221320 ~
16547 2460565661129971310275869112846043251843432691552928458573495971 ~
16548 5042565399302112184947232132380516549802909919676815118022483192 ~
16549 5127372199792134331067642187484426215985121676396779352982985195 ~
16550 8545392106957880586853123277545433229161989053189053725391582222 ~
16551 923259727813342781825606488233760719681014481453198336237910767 ~
16552 1255017528826351836492103572587410356573894694875444694018175923 ~
16553 0609370828146501857425324969212764624247832210765473750568198834 ~
16554 5641035458027261252285503154325039591848918982630498759115406321 ~
16555 0354263890012837426155187877318375862355175378506956599570028011 ~
16556 5841258870150030170259167463020842412449128392380525772514737141 ~
16557 2310230172563968305553583262840383638157686828464330456805994018 ~
16558 7001071952092970177990583216417579868116586547147748964716547948 ~
16559 8312140431836079844314055731179349677763739898930227765607058530 ~
16560 4083747752640947435070395214524701683884070908706147194437225650 ~
16561 2823145872995869738316897126851939042297110721350756978037262545 ~
16562 8141095038270388987364516284820180468288205829135339013835649144 ~
16563 3004015706509887926715417450706686888783438055583501196745862340 ~
16564 8059532724727843829259395771584036885940989939255241688378793572 ~
16565 7967951654076673927031256418760962190243046993485989199060012977 ~
16566 7469214532970421677817261517850653008552559997940209969455431545 ~
16567 2745856704403686680428648404512881182309793496962721836492935516 ~
16568 2029872469583299481932978335803459023227052612542114437084359584 ~
16569 944338363838317751841160881711251279233374577219339820819005406 ~
16570 3292937775306906607415304997682647124407768817248673421685881509 ~
16571 9133422075930947173855159340808957124410634720893194912880783576 ~
16572 3115829400549708918023366596077070927599010527028150868897828549 ~
16573 4340372642729262103487013992868853550062061514343078665396085995 ~
16574 0058714939141652065302070085265624074703660736605333805263766757 ~
16575 2018839497277047222153633851135483463624619855425993871933367482 ~
16576 0422097449956672702505446423243957506869591330193746919142980999 ~
16577 3424230550172665212092414559625960554427590951996824313084279693 ~
16578 7113207021049823238195747175985519501864630940297594363194450091 ~
16579 9150616049228764323192129703446093584259267276386814363309856853 ~
16580 2786024332141052330760658841495858718197071242995959226781172796 ~
16581 4438853796763139274314227953114500064922126500133268623021550837 ~
16582 \cs_end:
16583 }

```

(End definition for _fp_trig_inverse_two_pi:.)

_fp_trig_large:ww The exponent #1 is between 1 and 10000. We discard the integer part of $10^{\#1-16}/(2\pi)$,
_fp_trig_large_auxi:wwwwww that is, the first #1 digits of $10^{-16}/(2\pi)$, because it yields an integer contribution to
_fp_trig_large_auxii:ww
_fp_trig_large_auxiii:wnnnnnnnn
_fp_trig_large_auxiv:wN

$x/(2\pi)$. The `auxii` auxiliary discards 64 digits at a time thanks to spaces inserted in the result of `__fp_trig_inverse_two_pi:`, while `auxiii` discards 8 digits at a time, and `auxiv` discards digits one at a time. Then 64 digits are packed into groups of 4 and the `auxv` auxiliary is called.

```

16584 \cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6;
16585 {
16586   \exp_after:wN \__fp_trig_large_auxi:wwwww
16587   \int_value:w \__fp_int_eval:w (#1 - 32) / 64 \exp_after:wN ,
16588   \int_value:w \__fp_int_eval:w (#1 - 4) / 8 \exp_after:wN ,
16589   \int_value:w #1 \__fp_trig_inverse_two_pi: ;
16590   {#2}{#3}{#4}{#5} ;
16591 }
16592 \cs_new:Npn \__fp_trig_large_auxi:wwwww #1, #2, #3, #4!
16593 {
16594   \prg_replicate:nn {#1} { \__fp_trig_large_auxii:ww }
16595   \prg_replicate:nn { #2 - #1 * 8 }
16596     { \__fp_trig_large_auxiii:wNNNNNNNN }
16597   \prg_replicate:nn { #3 - #2 * 8 }
16598     { \__fp_trig_large_auxiv:wN }
16599   \prg_replicate:nn { 8 } { \__fp_pack_twice_four:wNNNNNNNN }
16600   \__fp_trig_large_auxv:www
16601 ;
16602 }
16603 \cs_new:Npn \__fp_trig_large_auxii:ww #1; #2 ~ { #1; }
16604 \cs_new:Npn \__fp_trig_large_auxiii:wNNNNNNNN
16605   #1; #2#3#4#5#6#7#8#9 { #1; }
16606 \cs_new:Npn \__fp_trig_large_auxiv:wN #1; #2 { #1; }

```

(End definition for `__fp_trig_large:ww` and others.)

```

\__fp_trig_large_auxv:www
\__fp_trig_large_auxvi:wNNNNNNNN
\__fp_trig_large_pack:NNNNw

```

First come the first 64 digits of the fractional part of $10^{*1-16}/(2\pi)$, arranged in 16 blocks of 4, and ending with a semicolon. Then some more digits of the same fractional part, ending with a semicolon, then 4 blocks of 4 digits holding the significand of the original argument. Multiply the 16-digit significand with the 64-digit fractional part: the `auxvi` auxiliary receives the significand as `#2#3#4#5` and 16 digits of the fractional part as `#6#7#8#9`, and computes one step of the usual ladder of `pack` functions we use for multiplication (see *e.g.*, `__fp_fixed_mul:wN`), then discards one block of the fractional part to set things up for the next step of the ladder. We perform 13 such steps, replacing the last `middle` shift by the appropriate `trailing` shift, then discard the significand and remaining 3 blocks from the fractional part, as there are not enough digits to compute any more step in the ladder. The last semicolon closes the ladder, and we return control to the `auxvii` auxiliary.

```

16607 \cs_new:Npn \__fp_trig_large_auxv:www #1; #2; #3;
16608 {
16609   \exp_after:wN \__fp_use_i_until_s:nw
16610   \exp_after:wN \__fp_trig_large_auxvii:w
16611   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
16612   \prg_replicate:nn { 13 }
16613     { \__fp_trig_large_auxvi:wNNNNNNNN }
16614     + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
16615   \__fp_use_i_until_s:nw
16616   ; #3 #1 ; ;
16617 }

```

```

16618 \cs_new:Npn \__fp_trig_large_auxvi:wnnnnnnnn #1; #2#3#4#5#6#7#8#9
16619 {
16620   \exp_after:wN \__fp_trig_large_pack:NNNNNw
16621   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
16622     + #2*#9 + #3*#8 + #4*#7 + #5*#6
16623     #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
16624 }
16625 \cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
16626 { + #1#2#3#4#5 ; #6 }

```

(End definition for __fp_trig_large_auxv:www, __fp_trig_large_auxvi:wnnnnnnnn, and __fp_trig_large_pack:NNNNNw.)

```

\__fp_trig_large_auxvii:w
\__fp_trig_large_auxviii:w
\__fp_trig_large_auxix:Nw
\__fp_trig_large_auxx:wNNNNN
\__fp_trig_large_auxxi:w

```

The `auxvii` auxiliary is followed by 52 digits and a semicolon. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of $\#1\#2\#3/125$, and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last `middle` shift is converted to a `trailing` shift. Any integer part (including negative values which come up when the octant is odd) is discarded by `__fp_use_i_until_s:nw`. The resulting fractional part should then be converted to radians by multiplying by $2\pi/8$, but first, build an extended precision number by abusing `__fp_ep_to_ep_loop:N` with the appropriate trailing markers. Finally, `__fp_trig_small:ww` sets up the argument for the functions which compute the Taylor series.

```

16627 \cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
16628 {
16629   \exp_after:wN \__fp_trig_large_auxviii:ww
16630   \int_value:w \__fp_int_eval:w (#1#2#3 - 62) / 125 ;
16631   #1#2#3
16632 }
16633 \cs_new:Npn \__fp_trig_large_auxviii:ww #1;
16634 {
16635   + #1
16636   \if_int_odd:w #1 \exp_stop_f:
16637     \exp_after:wN \__fp_trig_large_auxix:Nw
16638     \exp_after:wN -
16639   \else:
16640     \exp_after:wN \__fp_trig_large_auxix:Nw
16641     \exp_after:wN +
16642   \fi:
16643 }
16644 \cs_new:Npn \__fp_trig_large_auxix:Nw
16645 {
16646   \exp_after:wN \__fp_use_i_until_s:nw
16647   \exp_after:wN \__fp_trig_large_auxxi:w
16648   \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
16649   \prg_replicate:nn { 13 }
16650   { \__fp_trig_large_auxx:wNNNNN }
16651   + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
16652   ;
16653 }
16654 \cs_new:Npn \__fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
16655 {
16656   \exp_after:wN \__fp_trig_large_pack:NNNNNw
16657   \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int

```

```

16658      #2 8 * #3#4#5#6
16659      #1; #2
16660    }
16661 \cs_new:Npn \__fp_trig_large_auxxi:w #1;
16662 {
16663   \exp_after:wN \__fp_ep_mul_raw:wwwN
16664   \int_value:w \__fp_int_eval:w 0 \__fp_ep_to_ep_loop:N #1 ; ; !
16665   0,{7853}{9816}{3397}{4483}{0961}{5661};
16666   \__fp_trig_small:ww
16667 }

```

(End definition for `__fp_trig_large_auxvii:w` and others.)

31.1.6 Computing the power series

`__fp_sin_series_o:NNwww` Here we receive a conversion function `__fp_ep_to_float_o:wwN` or `__fp_ep_inv_to_float_o:wwN`, a *sign* (0 or 2), a (non-negative) *octant* delimited by a dot, a *fixed point* number delimited by a semicolon, and an extended-precision number. The auxiliary receives:

- the conversion function `#1`;
- the final sign, which depends on the octant `#3` and the sign `#2`;
- the octant `#3`, which controls the series we use;
- the square `#4 * #4` of the argument as a fixed point number, computed with `__fp_fixed_mul:wwn`;
- the number itself as an extended-precision number.

If the octant is in $\{1, 2, 5, 6, \dots\}$, we are near an extremum of the function and we use the series

$$\cos(x) = 1 - x^2 \left(\frac{1}{2!} - x^2 \left(\frac{1}{4!} - x^2 \left(\dots \right) \right) \right).$$

Otherwise, the series

$$\sin(x) = x \left(1 - x^2 \left(\frac{1}{3!} - x^2 \left(\frac{1}{5!} - x^2 \left(\dots \right) \right) \right) \right)$$

is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and `__fp_sanitizew` checks for overflow and underflow.

```

16668 \cs_new:Npn \__fp_sin_series_o:NNwww #1#2#3. #4;
16669 {
16670   \__fp_fixed_mul:wwn #4; #4;
16671   {
16672     \exp_after:wN \__fp_sin_series_aux_o:NNwww
16673     \exp_after:wN #1
16674     \int_value:w
16675     \if_int_odd:w \__fp_int_eval:w (#3 + 2) / 4 \__fp_int_eval_end:
16676       #2
16677     \else:
16678       \if_meaning:w #2 0 2 \else: 0 \fi:
16679       \fi:
16680     {#3}

```

```

16681     }
16682   }
16683   \cs_new:Npn \__fp_sin_series_aux_o:NNnwww #1#2#3 #4; #5,#6;
16684   {
16685     \if_int_odd:w \__fp_int_eval:w #3 / 2 \__fp_int_eval_end:
16686       \exp_after:wN \use_i:nn
16687     \else:
16688       \exp_after:wN \use_ii:nn
16689     \fi:
16690     { % 1/18!
16691       \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0001}{5619}{2070};
16692       #4;{0000}{0000}{0000}{0477}{9477}{3324};
16693       \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
16694       \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
16695       \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
16696       \__fp_fixed_mul_sub_back:wwwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
16697       \__fp_fixed_mul_sub_back:wwwn #4;{0013}{8888}{8888}{8888}{8888}{8889};
16698       \__fp_fixed_mul_sub_back:wwwn #4;{0416}{6666}{6666}{6666}{6666}{6667};
16699       \__fp_fixed_mul_sub_back:wwwn #4;{5000}{0000}{0000}{0000}{0000}{0000};
16700       \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
16701       { \__fp_fixed_continue:wn 0, }
16702     }
16703     { % 1/17!
16704       \__fp_fixed_mul_sub_back:wwwn {0000}{0000}{0000}{0028}{1145}{7254};
16705       #4;{0000}{0000}{0000}{7647}{1637}{3182};
16706       \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
16707       \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
16708       \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
16709       \__fp_fixed_mul_sub_back:wwwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
16710       \__fp_fixed_mul_sub_back:wwwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
16711       \__fp_fixed_mul_sub_back:wwwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
16712       \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
16713       { \__fp_ep_mul:wwwn 0, } #5,#6;
16714     }
16715     {
16716       \exp_after:wN \__fp_sanitize:Nw
16717       \exp_after:wN #2
16718       \int_value:w \__fp_int_eval:w #1
16719     }
16720     #2
16721   }

```

(End definition for __fp_sin_series_o:NNnwww and __fp_sin_series_aux_o:NNnwww.)

__fp_tan_series_o:NNnwww Contrarily to __fp_sin_series_o:NNnwww which received a conversion auxiliary as #1, here, #1 is 0 for tangent and 2 for cotangent. Consider first the case of the tangent. The octant #3 starts at 1, which means that it is 1 or 2 for $|x| \in [0, \pi/2]$, it is 3 or 4 for $|x| \in [\pi/2, \pi]$, and so on: the intervals on which $\tan|x| \geq 0$ coincide with those for which $\lfloor (\#3 + 1)/2 \rfloor$ is odd. We also have to take into account the original sign of x to get the sign of the final result; it is straightforward to check that the first \int_value:w expansion produces 0 for a positive final result, and 2 otherwise. A similar story holds for $\cot(x)$.

The auxiliary receives the sign, the octant, the square of the (reduced) input, and the (reduced) input (an extended-precision number) as arguments. It then computes the

numerator and denominator of

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2 a_5))))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2 b_5)))}.$$

The ratio is computed by `__fp_ep_div:wwwn`, then converted to a floating point number. For octants #3 (really, quadrants) next to a pole of the functions, the fixed point numerator and denominator are exchanged before computing the ratio. Note that this `\if_int_odd:w` test relies on the fact that the octant is at least 1.

```

16722 \cs_new:Npn \__fp_tan_series_o:NNwww #1#2#3. #4;
16723 {
16724   \__fp_fixed_mul:wwn #4; #4;
16725   {
16726     \exp_after:wN \__fp_tan_series_aux_o:Nnwww
16727     \int_value:w
16728     \if_int_odd:w \__fp_int_eval:w #3 / 2 \__fp_int_eval_end:
16729     \exp_after:wN \reverse_if:N
16730     \fi:
16731     \if_meaning:w #1#2 2 \else: 0 \fi:
16732     {#3}
16733   }
16734 }
16735 \cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
16736 {
16737   \__fp_fixed_mul_sub_back:wwn {0000}{0000}{1527}{3493}{0856}{7059};
16738   #3; {0000}{0159}{6080}{0274}{5257}{6472};
16739   \__fp_fixed_mul_sub_back:wwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
16740   \__fp_fixed_mul_sub_back:wwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
16741   \__fp_fixed_mul_sub_back:wwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
16742   \__fp_fixed_mul_sub_back:wwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
16743   { \__fp_ep_mul:wwwn 0, } #4,#5;
16744   {
16745     \__fp_fixed_mul_sub_back:wwn {0000}{0007}{0258}{0681}{9408}{4706};
16746     #3; {0000}{2343}{7175}{1399}{6151}{7670};
16747     \__fp_fixed_mul_sub_back:wwn #3; {0019}{2638}{4588}{9232}{8861}{3691};
16748     \__fp_fixed_mul_sub_back:wwn #3; {0536}{6357}{0691}{4344}{6852}{4252};
16749     \__fp_fixed_mul_sub_back:wwn #3; {5263}{1578}{9473}{6842}{1052}{6315};
16750     \__fp_fixed_mul_sub_back:wwn #3; {10000}{0000}{0000}{0000}{0000}{0000};
16751     {
16752       \reverse_if:N \if_int_odd:w
16753       \__fp_int_eval:w (#2 - 1) / 2 \__fp_int_eval_end:
16754       \exp_after:wN \__fp_reverse_args:Nww
16755       \fi:
16756       \__fp_ep_div:wwwn 0,
16757     }
16758   }
16759   {
16760     \exp_after:wN \__fp_sanitise:Nw
16761     \exp_after:wN #1
16762     \int_value:w \__fp_int_eval:w \__fp_ep_to_float_o:wwN
16763   }
16764   #1
16765 }

```

(End definition for `__fp_tan_series_o:NNwww` and `__fp_tan_series_aux_o:Nnwww`.)

31.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arcsecant, and arcsecant) are based on a function often denoted **atan2**. This function is accessed directly by feeding two arguments to arctangent, and is defined by $\text{atan}(y, x) = \text{atan}(y/x)$ for generic y and x . Its advantages over the conventional arctangent is that it takes values in $[-\pi, \pi]$ rather than $[-\pi/2, \pi/2]$, and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of **atan** as

$$\text{acos } x = \text{atan}(\sqrt{1-x^2}, x) \quad (5)$$

$$\text{asin } x = \text{atan}(x, \sqrt{1-x^2}) \quad (6)$$

$$\text{asec } x = \text{atan}(\sqrt{x^2-1}, 1) \quad (7)$$

$$\text{acsc } x = \text{atan}(1, \sqrt{x^2-1}) \quad (8)$$

$$\text{atan } x = \text{atan}(x, 1) \quad (9)$$

$$\text{acot } x = \text{atan}(1, x). \quad (10)$$

Rather than introducing a new function, **atan2**, the arctangent function **atan** is overloaded: it can take one or two arguments. In the comments below, following many texts, we call the first argument y and the second x , because $\text{atan}(y, x) = \text{atan}(y/x)$ is the angular coordinate of the point (x, y) .

As for direct trigonometric functions, the first step in computing $\text{atan}(y, x)$ is argument reduction. The sign of y gives that of the result. We distinguish eight regions where the point $(x, |y|)$ can lie, of angular size roughly $\pi/8$, characterized by their “octant”, between 0 and 7 included. In each region, we compute an arctangent as a Taylor series, then shift this arctangent by the appropriate multiple of $\pi/4$ and sign to get the result. Here is a list of octants, and how we compute the arctangent (we assume $y > 0$: otherwise replace y by $-y$ below):

0 $0 < |y| < 0.41421x$, then $\text{atan } \frac{|y|}{x}$ is given by a nicely convergent Taylor series;

1 $0 < 0.41421x < |y| < x$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{4} - \text{atan } \frac{x-|y|}{x+|y|}$;

2 $0 < 0.41421|y| < x < |y|$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{4} + \text{atan } \frac{-x+|y|}{x+|y|}$;

3 $0 < x < 0.41421|y|$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{2} - \text{atan } \frac{x}{|y|}$;

4 $0 < -x < 0.41421|y|$, then $\text{atan } \frac{|y|}{x} = \frac{\pi}{2} + \text{atan } \frac{-x}{|y|}$;

5 $0 < 0.41421|y| < -x < |y|$, then $\text{atan } \frac{|y|}{x} = \frac{3\pi}{4} - \text{atan } \frac{x+|y|}{-x+|y|}$;

6 $0 < -0.41421x < |y| < -x$, then $\text{atan } \frac{|y|}{x} = \frac{3\pi}{4} + \text{atan } \frac{-x-|y|}{-x+|y|}$;

7 $0 < |y| < -0.41421x$, then $\text{atan } \frac{|y|}{x} = \pi - \text{atan } \frac{|y|}{-x}$.

In the following, we denote by z the ratio among $|\frac{y}{x}|$, $|\frac{x}{y}|$, $|\frac{x+y}{x-y}|$, $|\frac{x-y}{x+y}|$ which appears in the right-hand side above.

31.2.1 Arctangent and arccotangent

`__fp_atan_o:Nw` The parsing step manipulates `atan` and `acot` like `min` and `max`, reading in an array of
`__fp_acot_o:Nw` operands, but also leaves `\use_i:nn` or `\use_ii:nn` depending on whether the result
`__fp_atan_default:w` should be given in radians or in degrees. The helper `__fp_parse_function_one_`
`two:nnw` checks that the operand is one or two floating point numbers (not tuples) and
leaves its second argument or its tail accordingly (its first argument is used for error
messages). More precisely if we are given a single floating point number `__fp_atan_`
`default:w` places `\c_one_fp` (expanded) after it; otherwise `__fp_atan_default:w` is
omitted by `__fp_parse_function_one_two:nnw`.

```

16766 \cs_new:Npn \__fp_atan_o:Nw #1
16767 {
16768   \__fp_parse_function_one_two:nnw
16769   { #1 { atan } { atand } }
16770   { \__fp_atan_default:w \__fp_atanii_o:Nww #1 }
16771 }
16772 \cs_new:Npn \__fp_acot_o:Nw #1
16773 {
16774   \__fp_parse_function_one_two:nnw
16775   { #1 { acot } { acotd } }
16776   { \__fp_atan_default:w \__fp_acotii_o:Nww #1 }
16777 }
16778 \cs_new:Npx \__fp_atan_default:w #1#2#3 @ { #1 #2 #3 \c_one_fp @ }

```

(End definition for `__fp_atan_o:Nw`, `__fp_acot_o:Nw`, and `__fp_atan_default:w`.)

`__fp_atanii_o:Nww` If either operand is `nan`, we return it. If both are normal, we call `__fp_atan_normal_`
`__fp_acotii_o:Nww` `o:NNnwNnw`. If both are zero or both infinity, we call `__fp_atan_inf_o:NNNw` with
argument 2, leading to a result among $\{\pm\pi/4, \pm3\pi/4\}$ (in degrees, $\{\pm45, \pm135\}$). Oth-
erwise, one is much bigger than the other, and we call `__fp_atan_inf_o:NNNw` with
either an argument of 4, leading to the values $\pm\pi/2$ (in degrees, ±90), or 0, leading to
 $\{\pm0, \pm\pi\}$ (in degrees, $\{\pm0, \pm180\}$). Since `acot(x,y) = atan(y,x)`, `__fp_acotii_o:ww`
simply reverses its two arguments.

```

16779 \cs_new:Npn \__fp_atanii_o:Nww
16780 #1 \s__fp \__fp_chk:w #2#3#4; \s__fp \__fp_chk:w #5 #6 @
16781 {
16782   \if_meaning:w 3 #2 \__fp_case_return_i_o:ww \fi:
16783   \if_meaning:w 3 #5 \__fp_case_return_ii_o:ww \fi:
16784   \if_case:w
16785     \if_meaning:w #2 #5
16786     \if_meaning:w 1 #2 10 \else: 0 \fi:
16787   \else:
16788     \if_int_compare:w #2 > #5 \exp_stop_f: 1 \else: 2 \fi:
16789   \fi:
16790   \exp_stop_f:
16791   \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 2 }
16792   \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 4 }
16793   \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 0 }
16794   \fi:
16795   \__fp_atan_normal_o:NNnwNnw #1
16796   \s__fp \__fp_chk:w #2#3#4;
16797   \s__fp \__fp_chk:w #5 #6
16798 }

```

```

16799 \cs_new:Npn \__fp_acotii_o:Nww #1#2; #3;
16800 { \__fp_atanii_o:Nww #1#3; #2; }

```

(End definition for __fp_atanii_o:Nww and __fp_acotii_o:Nww.)

__fp_atan_inf_o:NNNw This auxiliary is called whenever one number is ± 0 or $\pm\infty$ (and neither is NaN). Then the result only depends on the signs, and its value is a multiple of $\pi/4$. We use the same auxiliary as for normal numbers, __fp_atan_combine_o:NwwwwwN, with arguments the final sign #2; the octant #3; $\operatorname{atan} z/z = 1$ as a fixed point number; $z = 0$ as a fixed point number; and $z = 0$ as an extended-precision number. Given the values we provide, $\operatorname{atan} z$ is computed to be 0, and the result is $[\#3/2] \cdot \pi/4$ if the sign #5 of x is positive, and $[(7 - \#3)/2] \cdot \pi/4$ for negative x , where the divisions are rounded up.

```

16801 \cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
16802 {
16803   \exp_after:wN \__fp_atan_combine_o:NwwwwwN
16804   \exp_after:wN #2
16805   \int_value:w \__fp_int_eval:w
16806   \if_meaning:w 2 #5 7 - \fi: #3 \exp_after:wN ;
16807   \c__fp_one_fixed_tl
16808   {0000}{0000}{0000}{0000}{0000}{0000};
16809   0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
16810 }

```

(End definition for __fp_atan_inf_o:NNNw.)

__fp_atan_normal_o:NNnwNnw Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like $\operatorname{atan}(x, \sqrt{1 - x^2})$ without intermediate rounding errors.

```

16811 \cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
16812 #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
16813 {
16814   \__fp_atan_test_o:NwwNwwN
16815   #2 #3, #4{0000}{0000};
16816   #5 #6, #7{0000}{0000}; #1
16817 }

```

(End definition for __fp_atan_normal_o:NNnwNnw.)

__fp_atan_test_o:NwwNwwN This receives: the sign #1 of y , its exponent #2, its 24 digits #3 in groups of 4, and similarly for x . We prepare to call __fp_atan_combine_o:NwwwwwN which expects the sign #1, the octant, the ratio $(\operatorname{atan} z)/z = 1 - \dots$, and the value of z , both as a fixed point number and as an extended-precision floating point number with a mantissa in $[0.01, 1)$. For now, we place #1 as a first argument, and start an integer expression for the octant. The sign of x does not affect z , so we simply leave a contribution to the octant: $\langle \text{octant} \rangle \rightarrow 7 - \langle \text{octant} \rangle$ for negative x . Then we order $|y|$ and $|x|$ in a non-decreasing order: if $|y| > |x|$, insert 3- in the expression for the octant, and swap the two numbers. The finer test with 0.41421 is done by __fp_atan_div:wnwnwnw after the operands have been ordered.

```

16818 \cs_new:Npn \__fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
16819 {
16820   \exp_after:wN \__fp_atan_combine_o:NwwwwwN

```

```

16821 \exp_after:wN #1
16822 \int_value:w \__fp_int_eval:w
16823 \if_meaning:w 2 #4
16824 7 - \__fp_int_eval:w
16825 \fi:
16826 \if_int_compare:w
16827 \__fp_ep_compare:www #2,#3; #5,#6; > 0 \exp_stop_f:
16828 3 -
16829 \exp_after:wN \__fp_reverse_args:Nww
16830 \fi:
16831 \__fp_atan_div:wnwnw #2,#3; #5,#6;
16832 }

```

(End definition for __fp_atan_test_o:NwwNwwN.)

__fp_atan_div:wnwnw
__fp_atan_near:www
__fp_atan_near_aux:wn

This receives two positive numbers a and b (equal to $|x|$ and $|y|$ in some order), each as an exponent and 6 blocks of 4 digits, such that $0 < a < b$. If $0.41421b < a$, the two numbers are “near”, hence the point (y, x) that we started with is closer to the diagonals $\{|y| = |x|\}$ than to the axes $\{xy = 0\}$. In that case, the octant is 1 (possibly combined with the 7- and 3- inserted earlier) and we wish to compute $\operatorname{atan} \frac{b-a}{a+b}$. Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute $\operatorname{atan} \frac{a}{b}$. In any case, call __fp_atan_auxi:ww followed by z , as a comma-delimited exponent and a fixed point number.

```

16833 \cs_new:Npn \__fp_atan_div:wnwnw #1,#2#3; #4,#5#6;
16834 {
16835 \if_int_compare:w
16836 \__fp_int_eval:w 41421 * #5 < #2 000
16837 \if_case:w \__fp_int_eval:w #4 - #1 \__fp_int_eval_end:
16838 00 \or: 0 \fi:
16839 \exp_stop_f:
16840 \exp_after:wN \__fp_atan_near:www
16841 \fi:
16842 0
16843 \__fp_ep_div:www #1,{#2}#3; #4,{#5}#6;
16844 \__fp_atan_auxi:ww
16845 }
16846 \cs_new:Npn \__fp_atan_near:www
16847 0 \__fp_ep_div:www #1,#2; #3,
16848 {
16849 1
16850 \__fp_ep_to_fixed:wn #1 - #3, #2;
16851 \__fp_atan_near_aux:wn
16852 }
16853 \cs_new:Npn \__fp_atan_near_aux:wn #1; #2;
16854 {
16855 \__fp_fixed_add:wn #1; #2;
16856 { \__fp_fixed_sub:wn #2; #1; { \__fp_ep_div:www 0, } 0, }
16857 }

```

(End definition for __fp_atan_div:wnwnw, __fp_atan_near:www, and __fp_atan_near_aux:wn.)

__fp_atan_auxi:ww
__fp_atan_auxii:w

Convert z from a representation as an exponent and a fixed point number in $[0.01, 1)$ to a fixed point number only, then set up the call to __fp_atan_Taylor_loop:www, followed by the fixed point representation of z and the old representation.

```

16858 \cs_new:Npn \__fp_atan_auxi:ww #1,#2;
16859 { \__fp_ep_to_fixed:wwn #1,#2; \__fp_atan_auxii:w #1,#2; }
16860 \cs_new:Npn \__fp_atan_auxii:w #1;
16861 {
16862   \__fp_fixed_mul:wwn #1; #1;
16863   {
16864     \__fp_atan_Taylor_loop:www 39 ;
16865     {0000}{0000}{0000}{0000}{0000}{0000} ;
16866   }
16867   ! #1;
16868 }

```

(End definition for __fp_atan_auxi:ww and __fp_atan_auxii:w.)

__fp_atan_Taylor_loop:www
 __fp_atan_Taylor_break:w

We compute the series of $(\operatorname{atan} z)/z$. A typical intermediate stage has $\#1 = 2k - 1$, $\#2 = \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\dots - z^2\frac{1}{39}))$, and $\#3 = z^2$. To go to the next step $k \rightarrow k - 1$, we compute $\frac{1}{2k-1}$, then subtract from it z^2 times $\#2$. The loop stops when $k = 0$: then $\#2$ is $(\operatorname{atan} z)/z$, and there is a need to clean up all the unnecessary data, end the integer expression computing the octant with a semicolon, and leave the result $\#2$ afterwards.

```

16869 \cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
16870 {
16871   \if_int_compare:w #1 = -1 \exp_stop_f:
16872   \__fp_atan_Taylor_break:w
16873   \fi:
16874   \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl #1;
16875   \__fp_rrot:www \__fp_fixed_mul_sub_back:wwn #2; #3;
16876   {
16877     \exp_after:wN \__fp_atan_Taylor_loop:www
16878     \int_value:w \__fp_int_eval:w #1 - 2 ;
16879   }
16880   #3;
16881 }
16882 \cs_new:Npn \__fp_atan_Taylor_break:w
16883   \fi: #1 \__fp_fixed_mul_sub_back:wwn #2; #3 !
16884   { \fi: ; #2 ; }

```

(End definition for __fp_atan_Taylor_loop:www and __fp_atan_Taylor_break:w.)

__fp_atan_combine_o:NwwwwwN
 __fp_atan_combine_aux:ww

This receives a $\langle sign \rangle$, an $\langle octant \rangle$, a fixed point value of $(\operatorname{atan} z)/z$, a fixed point number z , and another representation of z , as an $\langle exponent \rangle$ and the fixed point number $10^{-\langle exponent \rangle} z$, followed by either `\use_i:nn` (when working in radians) or `\use_ii:nn` (when working in degrees). The function computes the floating point result

$$\langle sign \rangle \left(\left\lceil \frac{\langle octant \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant \rangle} \frac{\operatorname{atan} z}{z} \cdot z \right), \quad (11)$$

multiplied by $180/\pi$ if working in degrees, and using in any case the most appropriate representation of z . The floating point result is passed to `__fp_sanitize:Nw`, which checks for overflow or underflow. If the octant is 0, leave the exponent $\#5$ for `__fp_sanitize:Nw`, and multiply $\#3 = \frac{\operatorname{atan} z}{z}$ with $\#6$, the adjusted z . Otherwise, multiply $\#3 = \frac{\operatorname{atan} z}{z}$ with $\#4 = z$, then compute the appropriate multiple of $\frac{\pi}{4}$ and add or subtract the product $\#3 \cdot \#4$. In both cases, convert to a floating point with `__fp_fixed_to_float_o:wN`.

```

16885 \cs_new:Npn \__fp_atan_combine_o:NwwwwwN #1 #2; #3; #4; #5,#6; #7
16886 {
16887   \exp_after:wN \__fp_sanitizew
16888   \exp_after:wN #1
16889   \int_value:w \__fp_int_eval:w
16890   \if_meaning:w 0 #2
16891     \exp_after:wN \use_i:nn
16892   \else:
16893     \exp_after:wN \use_ii:nn
16894   \fi:
16895   { #5 \__fp_fixed_mul:wwn #3; #6; }
16896   {
16897     \__fp_fixed_mul:wwn #3; #4;
16898     {
16899       \exp_after:wN \__fp_atan_combine_aux:ww
16900       \int_value:w \__fp_int_eval:w #2 / 2 ; #2;
16901     }
16902   }
16903   { #7 \__fp_fixed_to_float_o:wN \__fp_fixed_to_float_rad_o:wN }
16904   #1
16905 }
16906 \cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
16907 {
16908   \__fp_fixed_mul_short:wwn
16909   {7853}{9816}{3397}{4483}{0961}{5661};
16910   {#1}{0000}{0000};
16911   {
16912     \if_int_odd:w #2 \exp_stop_f:
16913     \exp_after:wN \__fp_fixed_sub:wwn
16914   \else:
16915     \exp_after:wN \__fp_fixed_add:wwn
16916   \fi:
16917 }
16918 }

```

(End definition for __fp_atan_combine_o:NwwwwwN and __fp_atan_combine_aux:ww.)

31.2.2 Arcsine and arccosine

__fp_asin_o:w Again, the first argument provided by l3fp-parse is \use_i:nn if we are to work in radians and \use_ii:nn for degrees. Then comes a floating point number. The arcsine of ± 0 or NaN is the same floating point number. The arcsine of $\pm\infty$ raises an invalid operation exception. Otherwise, call an auxiliary common with __fp_acos_o:w, feeding it information about what function is being performed (for “invalid operation” exceptions).

```

16919 \cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
16920 {
16921   \if_case:w #2 \exp_stop_f:
16922     \__fp_case_return_same_o:w
16923   \or:
16924     \__fp_case_use:nw
16925     { \__fp_asin_normal_o:NfwNnnnw #1 { #1 { asin } { asind } } }
16926   \or:
16927     \__fp_case_use:nw
16928     { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }

```

```

16929     \else:
16930         \__fp_case_return_same_o:w
16931     \fi:
16932     \s__fp \__fp_chk:w #2 #3;
16933 }

```

(End definition for __fp_asin_o:w.)

__fp_acos_o:w The arccosine of ± 0 is $\pi/2$ (in degrees, 90). The arccosine of $\pm\infty$ raises an invalid operation exception. The arccosine of NaN is itself. Otherwise, call an auxiliary common with __fp_sin_o:w, informing it that it was called by acos or acosd, and preparing to swap some arguments down the line.

```

16934 \cs_new:Npn \__fp_acos_o:w #1 \s__fp \__fp_chk:w #2#3; @
16935 {
16936     \if_case:w #2 \exp_stop_f:
16937         \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
16938     \or:
16939         \__fp_case_use:nw
16940         {
16941             \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
16942             \__fp_reverse_args:Nww
16943         }
16944     \or:
16945         \__fp_case_use:nw
16946         { \__fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
16947     \else:
16948         \__fp_case_return_same_o:w
16949     \fi:
16950     \s__fp \__fp_chk:w #2 #3;
16951 }

```

(End definition for __fp_acos_o:w.)

__fp_asin_normal_o:NfwNnnnnw If the exponent #5 is at most 0, the operand lies within $(-1, 1)$ and the operation is permitted: call __fp_asin_auxi_o:NnNww with the appropriate arguments. If the number is exactly ± 1 (the test works because we know that $\#5 \geq 1$, $\#6\#7 \geq 10000000$, $\#8\#9 \geq 0$, with equality only for ± 1), we also call __fp_asin_auxi_o:NnNww. Otherwise, __fp_use_i:ww gets rid of the asin auxiliary, and raises instead an invalid operation, because the operand is outside the domain of arcsine or arccosine.

```

16952 \cs_new:Npn \__fp_asin_normal_o:NfwNnnnnw
16953     #1#2#3 \s__fp \__fp_chk:w 1#4#5#6#7#8#9;
16954 {
16955     \if_int_compare:w #5 < 1 \exp_stop_f:
16956         \exp_after:wN \__fp_use_none_until_s:w
16957     \fi:
16958     \if_int_compare:w \__fp_int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
16959         \exp_after:wN \__fp_use_none_until_s:w
16960     \fi:
16961     \__fp_use_i:ww
16962     \__fp_invalid_operation_o:fw {#2}
16963     \s__fp \__fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9};
16964     \__fp_asin_auxi_o:NnNww
16965     #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
16966 }

```

(End definition for `_fp_asin_normal_o:NfwNnnnnw`.)

`_fp_asin_auxi_o:NnNww`
`_fp_asin_isqrt:wn`

We compute $x/\sqrt{1-x^2}$. This function is used by `asin` and `acos`, but also by `acsc` and `asec` after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate $1-x^2$ as $(1+x)(1-x)$: this behaves better near $x=1$. We do the addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root $1/\sqrt{1-x^2}$. Finally, multiply by the (positive) extended-precision float $|x|$, and feed the (signed) result, and the number $+1$, as arguments to the arctangent function. When computing the arccosine, the arguments $x/\sqrt{1-x^2}$ and $+1$ are swapped by `#2` (`_fp_reverse_args:Nww` in that case) before `_fp_atan_test_o:NwwNwwN` is evaluated. Note that the arctangent function requires normalized arguments, hence the need for `ep_to_ep` and `continue` after `ep_mul`.

```

16967 \cs_new:Npn \_fp_asin_auxi_o:NnNww #1#2#3#4,#5;
16968 {
16969   \_fp_ep_to_fixed:wn #4,#5;
16970   \_fp_asin_isqrt:wn
16971   \_fp_ep_mul:wwwwn #4,#5;
16972   \_fp_ep_to_ep:wwN
16973   \_fp_fixed_continue:wn
16974   { #2 \_fp_atan_test_o:NwwNwwN #3 }
16975   0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
16976 }
16977 \cs_new:Npn \_fp_asin_isqrt:wn #1;
16978 {
16979   \exp_after:wN \_fp_fixed_sub:wn \c__fp_one_fixed_tl #1;
16980   {
16981     \_fp_fixed_add_one:wn #1;
16982     \_fp_fixed_continue:wn { \_fp_ep_mul:wwwwn 0, } 0,
16983   }
16984   \_fp_ep_isqrt:wnn
16985 }

```

(End definition for `_fp_asin_auxi_o:NnNww` and `_fp_asin_isqrt:wn`.)

31.2.3 Arccosecant and arcsecant

`_fp_acsc_o:w`

Cases are mostly labelled by `#2`, except when `#2` is 2: then we use `#3#2`, which is $02=2$ when the number is $+\infty$ and 22 when the number is $-\infty$. The arccosecant of ± 0 raises an invalid operation exception. The arccosecant of $\pm\infty$ is ± 0 with the same sign. The arcosecant of NaN is itself. Otherwise, `_fp_acsc_normal_o:NfwNnw` does some more tests, keeping the function name (`acsc` or `acscd`) as an argument for invalid operation exceptions.

```

16986 \cs_new:Npn \_fp_acsc_o:w #1 \s__fp \_fp_chk:w #2#3#4; @
16987 {
16988   \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
16989     \_fp_case_use:nw
16990     { \_fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
16991   \or: \_fp_case_use:nw
16992     { \_fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
16993   \or: \_fp_case_return_o:Nw \c_zero_fp
16994   \or: \_fp_case_return_same_o:w
16995   \else: \_fp_case_return_o:Nw \c_minus_zero_fp

```



```

16996     \fi:
16997     \s__fp \__fp_chk:w #2 #3 #4;
16998   }

```

(End definition for __fp_acsc_o:w.)

__fp_asec_o:w The arcsecant of ± 0 raises an invalid operation exception. The arcsecant of $\pm\infty$ is $\pi/2$ (in degrees, 90). The arcosecant of NaN is itself. Otherwise, do some more tests, keeping the function name asec (or asecd) as an argument for invalid operation exceptions, and a __fp_reverse_args:Nww following precisely that appearing in __fp_acos_o:w.

```

16999 \cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
17000 {
17001   \if_case:w #2 \exp_stop_f:
17002     \__fp_case_use:nw
17003     { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
17004   \or:
17005     \__fp_case_use:nw
17006     {
17007       \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
17008       \__fp_reverse_args:Nww
17009     }
17010   \or: \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
17011   \else: \__fp_case_return_same_o:w
17012   \fi:
17013   \s__fp \__fp_chk:w #2 #3;
17014 }

```

(End definition for __fp_asec_o:w.)

__fp_acsc_normal_o:NfwNnw If the exponent is non-positive, the operand is less than 1 in absolute value, which is always an invalid operation: complain. Otherwise, compute the inverse of the operand, and feed it to __fp_asin_auxi_o:NnNww (with all the appropriate arguments). This computes what we want thanks to $\text{acsc}(x) = \text{asin}(1/x)$ and $\text{asec}(x) = \text{acos}(1/x)$.

```

17015 \cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
17016 {
17017   \int_compare:nNnTF {#5} < 1
17018   {
17019     \__fp_invalid_operation_o:fw {#2}
17020     \s__fp \__fp_chk:w 1#4{#5}#6;
17021   }
17022   {
17023     \__fp_ep_div:wwwwn
17024     1,{1000}{0000}{0000}{0000}{0000}{0000};
17025     #5,#6{0000}{0000};
17026     { \__fp_asin_auxi_o:NnNww #1 {#3} #4 }
17027   }
17028 }

```

(End definition for __fp_acsc_normal_o:NfwNnw.)

```

17029 </initex | package>

```

32 13fp-convert implementation

17030 $\langle *initex \mid package \rangle$

17031 $\langle @@=fp \rangle$

32.1 Dealing with tuples

The first argument is for instance `_fp_to_tl_dispatch:w`, which converts any floating point object to the appropriate representation. We loop through all items, putting ,~ between all of them and making sure to remove the leading ,~.

```

17032 \cs_new:Npn \_fp_tuple_convert:Nw #1 \s\_fp_tuple \_fp_tuple_chk:w #2 ;
17033 {
17034   \int_case:nnF { \_fp_array_count:n {#2} }
17035   {
17036     { 0 } { ( ) }
17037     { 1 } { \_fp_tuple_convert_end:w @ { #1 #2 , } }
17038   }
17039   {
17040     \_fp_tuple_convert_loop:nNw { } #1
17041     #2 { ? \_fp_tuple_convert_end:w } ;
17042     @ { \use_none:nn }
17043   }
17044 }
17045 \cs_new:Npn \_fp_tuple_convert_loop:nNw #1#2#3#4; #5 @ #6
17046 {
17047   \use_none:n #3
17048   \exp_args:Nf \_fp_tuple_convert_loop:nNw { #2 #3#4 ; } #2 #5
17049   @ { #6 , ~ #1 }
17050 }
17051 \cs_new:Npn \_fp_tuple_convert_end:w #1 @ #2
17052 { \exp_after:wN ( \exp:w \exp_end_continue_f:w #2 ) }

```

(End definition for `_fp_tuple_convert:Nw`, `_fp_tuple_convert_loop:nNw`, and `_fp_tuple_convert_end:w`.)

32.2 Trimming trailing zeros

If #1 ends with a 0, the loop auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same loop auxiliary. Once the last trailing zero is reached, the second argument is the dot auxiliary, which removes a trailing dot if any. We then clean-up with the end auxiliary, keeping only the number.

```

17053 \cs_new:Npn \_fp_trim_zeros:w #1 ;
17054 {
17055   \_fp_trim_zeros_loop:w #1
17056   ; \_fp_trim_zeros_loop:w 0; \_fp_trim_zeros_dot:w .; \s_stop
17057 }
17058 \cs_new:Npn \_fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
17059 \cs_new:Npn \_fp_trim_zeros_dot:w #1 .; { \_fp_trim_zeros_end:w #1 ; }
17060 \cs_new:Npn \_fp_trim_zeros_end:w #1 ; #2 \s_stop { #1 }

```

(End definition for `_fp_trim_zeros:w` and others.)

32.3 Scientific notation

`\fp_to_scientific:N` The three public functions evaluate their argument, then pass it to `__fp_to_scientific_dispatch:w`.

```
\fp_to_scientific:c
\fp_to_scientific:n
17061 \cs_new:Npn \fp_to_scientific:N #1
17062 { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
17063 \cs_generate_variant:Nn \fp_to_scientific:N { c }
17064 \cs_new:Npn \fp_to_scientific:n
17065 {
17066   \exp_after:wN \__fp_to_scientific_dispatch:w
17067   \exp:w \exp_end_continue_f:w \__fp_parse:n
17068 }
```

(End definition for `\fp_to_scientific:N` and `\fp_to_scientific:n`. These functions are documented on page 181.)

`_fp_to_scientific_dispatch:w` We allow tuples.

```
\_fp_to_scientific_recover:w
\__fp_tuple_to_scientific:w
17069 \cs_new:Npn \__fp_to_scientific_dispatch:w #1
17070 {
17071   \__fp_change_func_type:NNN
17072   #1 \__fp_to_scientific:w \__fp_to_scientific_recover:w
17073   #1
17074 }
17075 \cs_new:Npn \__fp_to_scientific_recover:w #1 #2 ;
17076 {
17077   \__fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { } { }
17078   nan
17079 }
17080 \cs_new:Npn \__fp_tuple_to_scientific:w
17081 { \__fp_tuple_convert:Nw \__fp_to_scientific_dispatch:w }
```

(End definition for `__fp_to_scientific_dispatch:w`, `__fp_to_scientific_recover:w`, and `__fp_tuple_to_scientific:w`.)

`__fp_to_scientific:w` Expressing an internal floating point number in scientific notation is quite easy: no rounding, and the format is very well defined. First cater for the sign: negative numbers (`#2 = 2`) start with `-`; we then only need to care about positive numbers and `nan`. Then filter the special cases: ± 0 are represented as 0; infinities are converted to a number slightly larger than the largest after an “invalid_operation” exception; `nan` is represented as 0 after an “invalid_operation” exception. In the normal case, decrement the exponent and unbrace the 4 brace groups, then in a second step grab the first digit (previously hidden in braces) to order the various parts correctly.

```
17082 \cs_new:Npn \__fp_to_scientific:w \s__fp \__fp_chk:w #1#2
17083 {
17084   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
17085   \if_case:w #1 \exp_stop_f:
17086     \__fp_case_return:nw { 0.0000000000000000e0 }
17087   \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
17088   \or:
17089     \__fp_case_use:nw
17090     {
17091       \__fp_invalid_operation:nnw
17092       { \fp_to_scientific:N \c__fp_overflowing_fp }
17093       { fp_to_scientific }
17094     }
17095 }
```

```

17094     }
17095     \or:
17096     \__fp_case_use:nw
17097     {
17098         \__fp_invalid_operation:nnw
17099         { \fp_to_scientific:N \c_zero_fp }
17100         { fp_to_scientific }
17101     }
17102     \fi:
17103     \s__fp \__fp_chk:w #1 #2
17104 }
17105 \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
17106 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
17107 {
17108     \exp_after:wN \__fp_to_scientific_normal:wNw
17109     \exp_after:wN e
17110     \int_value:w \__fp_int_eval:w #2 - 1
17111     ; #3 #4 #5 #6 ;
17112 }
17113 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
17114 { #2.#3 #1 }

```

(End definition for `__fp_to_scientific:w`, `__fp_to_scientific_normal:wnnnnn`, and `__fp_to_scientific_normal:wNw`.)

32.4 Decimal representation

`\fp_to_decimal:N` All three public variants are based on the same `__fp_to_decimal_dispatch:w` after evaluating their argument to an internal floating point.

`\fp_to_decimal:c`

```

17115 \cs_new:Npn \fp_to_decimal:N #1
17116 { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
17117 \cs_generate_variant:Nn \fp_to_decimal:N { c }
17118 \cs_new:Npn \fp_to_decimal:n
17119 {
17120     \exp_after:wN \__fp_to_decimal_dispatch:w
17121     \exp:w \exp_end_continue_f:w \__fp_parse:n
17122 }

```

(End definition for `\fp_to_decimal:N` and `\fp_to_decimal:n`. These functions are documented on page 180.)

`__fp_to_decimal_dispatch:w` We allow tuples.

```

\__fp_to_decimal_recover:w
\__fp_tuple_to_decimal:w
17123 \cs_new:Npn \__fp_to_decimal_dispatch:w #1
17124 {
17125     \__fp_change_func_type:NNN
17126     #1 \__fp_to_decimal:w \__fp_to_decimal_recover:w
17127     #1
17128 }
17129 \cs_new:Npn \__fp_to_decimal_recover:w #1 #2 ;
17130 {
17131     \__fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
17132     nan
17133 }
17134 \cs_new:Npn \__fp_tuple_to_decimal:w
17135 { \__fp_tuple_convert:Nw \__fp_to_decimal_dispatch:w }

```

(End definition for `__fp_to_decimal_dispatch:w`, `__fp_to_decimal_recover:w`, and `__fp_tuple_to_decimal:w`.)

`__fp_to_decimal:w`
`__fp_to_decimal_normal:wnnnnn`
`__fp_to_decimal_large:Nnnw`
`__fp_to_decimal_huge:wnnnn`

The structure is similar to `__fp_to_scientific:w`. Insert - for negative numbers. Zero gives 0, $\pm\infty$ and NaN yield an “invalid operation” exception; note that $\pm\infty$ produces a very large output, which we don’t expand now since it most likely won’t be needed. Normal numbers with an exponent in the range [1,15] have that number of digits before the decimal separator: “decimate” them, and remove leading zeros with `\int_value:w`, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be 0.(zeros)<digits>, trimmed.

```

17136 \cs_new:Npn \__fp_to_decimal:w \s__fp \__fp_chk:w #1#2
17137 {
17138   \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
17139   \if_case:w #1 \exp_stop_f:
17140     \__fp_case_return:nw { 0 }
17141   \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
17142   \or:
17143     \__fp_case_use:nw
17144     {
17145       \__fp_invalid_operation:nnw
17146       { \fp_to_decimal:N \c__fp_overflowing_fp }
17147       { fp_to_decimal }
17148     }
17149   \or:
17150     \__fp_case_use:nw
17151     {
17152       \__fp_invalid_operation:nnw
17153       { 0 }
17154       { fp_to_decimal }
17155     }
17156   \fi:
17157   \s__fp \__fp_chk:w #1 #2
17158 }
17159 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
17160 \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
17161 {
17162   \int_compare:nNnTF {#2} > 0
17163   {
17164     \int_compare:nNnTF {#2} < \c__fp_prec_int
17165     {
17166       \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
17167       \__fp_to_decimal_large:Nnnw
17168     }
17169     {
17170       \exp_after:wN \exp_after:wN
17171       \exp_after:wN \__fp_to_decimal_huge:wnnnnn
17172       \prg_replicate:nn { #2 - \c__fp_prec_int } { 0 } ;
17173     }
17174     {#3} {#4} {#5} {#6}
17175   }
17176   {
17177     \exp_after:wN \__fp_trim_zeros:w
17178     \exp_after:wN 0

```

```

17179         \exp_after:wN .
17180         \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
17181         #3#4#5#6 ;
17182     }
17183 }
17184 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
17185 {
17186     \exp_after:wN \__fp_trim_zeros:w \int_value:w
17187     \if_int_compare:w #2 > 0 \exp_stop_f:
17188     #2
17189     \fi:
17190     \exp_stop_f:
17191     #3.#4 ;
17192 }
17193 \cs_new:Npn \__fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }

```

(End definition for `__fp_to_decimal:w` and others.)

32.5 Token list representation

`\fp_to_tl:N` These three public functions evaluate their argument, then pass it to `__fp_to_tl_dispatch:w`.
`\fp_to_tl:c`
`\fp_to_tl:n`

```

17194 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
17195 \cs_generate_variant:Nn \fp_to_tl:N { c }
17196 \cs_new:Npn \fp_to_tl:n
17197 {
17198     \exp_after:wN \__fp_to_tl_dispatch:w
17199     \exp:w \exp_end_continue_f:w \__fp_parse:n
17200 }

```

(End definition for `\fp_to_tl:N` and `\fp_to_tl:n`. These functions are documented on page 181.)

`__fp_to_tl_dispatch:w` We allow tuples.
`__fp_to_tl_recover:w`
`__fp_tuple_to_tl:w`

```

17201 \cs_new:Npn \__fp_to_tl_dispatch:w #1
17202 { \__fp_change_func_type:NNN #1 \__fp_to_tl:w \__fp_to_tl_recover:w #1 }
17203 \cs_new:Npn \__fp_to_tl_recover:w #1 #2 ;
17204 {
17205     \__fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
17206     nan
17207 }
17208 \cs_new:Npn \__fp_tuple_to_tl:w
17209 { \__fp_tuple_convert:Nw \__fp_to_tl_dispatch:w }

```

(End definition for `__fp_to_tl_dispatch:w`, `__fp_to_tl_recover:w`, and `__fp_tuple_to_tl:w`.)

`__fp_to_tl:w` A structure similar to `__fp_to_scientific_dispatch:w` and `__fp_to_decimal_dispatch:w`, but without the “invalid operation” exception. First filter special cases.
`__fp_to_tl_normal:nnnnn` We express normal numbers in decimal notation if the exponent is in the range $[-2, 16]$,
`__fp_to_tl_scientific:wnnnnn` and otherwise use scientific notation.
`__fp_to_tl_scientific:wNw`

```

17210 \cs_new:Npn \__fp_to_tl:w \s__fp \__fp_chk:w #1#2
17211 {
17212     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
17213     \if_case:w #1 \exp_stop_f:
17214         \__fp_case_return:nw { 0 }

```

```

17215     \or:   \exp_after:wN \__fp_to_tl_normal:nnnnn
17216     \or:   \__fp_case_return:nw { inf }
17217     \else: \__fp_case_return:nw { nan }
17218     \fi:
17219   }
17220 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
17221 {
17222   \int_compare:nTF
17223     { -2 <= #1 <= \c__fp_prec_int }
17224     { \__fp_to_decimal_normal:wnnnnn }
17225     { \__fp_to_tl_scientific:wnnnnn }
17226   \s__fp \__fp_chk:w 1 0 {#1}
17227 }
17228 \cs_new:Npn \__fp_to_tl_scientific:wnnnnn
17229   \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
17230 {
17231   \exp_after:wN \__fp_to_tl_scientific:wNw
17232   \exp_after:wN e
17233   \int_value:w \__fp_int_eval:w #2 - 1
17234   ; #3 #4 #5 #6 ;
17235 }
17236 \cs_new:Npn \__fp_to_tl_scientific:wNw #1 ; #2#3;
17237 { \__fp_trim_zeros:w #2.#3 ; #1 }

```

(End definition for `__fp_to_tl:w` and others.)

32.6 Formatting

This is not implemented yet, as it is not yet clear what a correct interface would be, for this kind of structured conversion from a floating point (or other types of variables) to a string. Ideas welcome.

32.7 Convert to dimension or integer

`\fp_to_dim:N` All three public variants are based on the same `__fp_to_dim_dispatch:w` after evaluating their argument to an internal floating point. We only allow floating point numbers, not tuples.

`\fp_to_dim:c`

`\fp_to_dim:n`

```

\__fp_to_dim_dispatch:w 17238 \cs_new:Npn \fp_to_dim:N #1
\__fp_to_dim_recover:w 17239 { \exp_after:wN \__fp_to_dim_dispatch:w #1 }
\__fp_to_dim:w 17240 \cs_generate_variant:Nn \fp_to_dim:N { c }
17241 \cs_new:Npn \fp_to_dim:n
17242 {
17243   \exp_after:wN \__fp_to_dim_dispatch:w
17244   \exp:w \exp_end_continue_f:w \__fp_parse:n
17245 }
17246 \cs_new:Npn \__fp_to_dim_dispatch:w #1#2 ;
17247 {
17248   \__fp_change_func_type:NNN #1 \__fp_to_dim:w \__fp_to_dim_recover:w
17249   #1 #2 ;
17250 }
17251 \cs_new:Npn \__fp_to_dim_recover:w #1
17252 { \__fp_invalid_operation:nnw { Opt } { fp_to_dim } }
17253 \cs_new:Npn \__fp_to_dim:w #1 ; { \__fp_to_decimal:w #1 ; pt }

```

(End definition for `\fp_to_dim:N` and others. These functions are documented on page 181.)

```

\fp_to_int:N For the most part identical to \fp_to_dim:N but without pt, and where \__fp_to_int:w
\fp_to_int:c does more work. To convert to an integer, first round to 0 places (to the nearest integer),
\fp_to_int:n then express the result as a decimal number: the definition of \__fp_to_decimal_-
\__fp_to_int_dispatch:w dispatch:w is such that there are no trailing dot nor zero.
\__fp_to_int_recover:w
17254 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
17255 \cs_generate_variant:Nn \fp_to_int:N { c }
17256 \cs_new:Npn \fp_to_int:n
17257 {
17258   \exp_after:wN \__fp_to_int_dispatch:w
17259   \exp:w \exp_end_continue_f:w \__fp_parse:n
17260 }
17261 \cs_new:Npn \__fp_to_int_dispatch:w #1#2 ;
17262 {
17263   \__fp_change_func_type:NNN #1 \__fp_to_int:w \__fp_to_int_recover:w
17264   #1 #2 ;
17265 }
17266 \cs_new:Npn \__fp_to_int_recover:w #1
17267 { \__fp_invalid_operation:nmw { 0 } { fp_to_int } }
17268 \cs_new:Npn \__fp_to_int:w #1;
17269 {
17270   \exp_after:wN \__fp_to_decimal:w \exp:w \exp_end_continue_f:w
17271   \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
17272 }

```

(End definition for `\fp_to_int:N` and others. These functions are documented on page 181.)

32.8 Convert from a dimension

`\dim_to_fp:n` The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (*i.e.*, expressed in scaled points), then multiplied by $2^{-16} = 0.0000152587890625$ to give a value expressed in points. The auxiliary `__fp_mul_npos_o:Nww` expects the desired *final sign* and two floating point operands (of the form `\s__fp ... ;`) as arguments. This set of functions is also used to convert dimension registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of `__fp_from_dim_test:ww`, and is combined with the exponent -4 of 2^{-16} . There is also a need to expand afterwards: this is performed by `__fp_mul_npos_o:Nww`, and cancelled by `\prg_do_nothing:` here.

```

17273 \__kernel_patch_args:nNNpn { { (#1) } }
17274 \cs_new:Npn \dim_to_fp:n #1
17275 {
17276   \exp_after:wN \__fp_from_dim_test:ww
17277   \exp_after:wN 0
17278   \exp_after:wN ,
17279   \int_value:w \etex_glueexpr:D #1 ;
17280 }
17281 \cs_new:Npn \__fp_from_dim_test:ww #1, #2
17282 {
17283   \if_meaning:w 0 #2
17284     \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
17285   \else:
17286     \exp_after:wN \__fp_from_dim:wNw

```



```

17287     \int_value:w \__fp_int_eval:w #1 - 4
17288     \if_meaning:w - #2
17289         \exp_after:wN , \exp_after:wN 2 \int_value:w
17290     \else:
17291         \exp_after:wN , \exp_after:wN 0 \int_value:w #2
17292     \fi:
17293 \fi:
17294 }
17295 \cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
17296 {
17297     \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
17298     #3 000 0000 00 {10}987654321; #2 {#1}
17299 }
17300 \cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
17301 { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
17302 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
17303 {
17304     \__fp_mul_npos_o:Nww #7
17305     \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
17306     \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
17307     \prg_do_nothing:
17308 }

```

(End definition for `\dim_to_fp:n` and others. This function is documented on page 157.)

32.9 Use and eval

\fp_use:N Those public functions are simple copies of the decimal conversions.
\fp_use:c 17309 \cs_new_eq:NN \fp_use:N \fp_to_decimal:N
\fp_eval:n 17310 \cs_generate_variant:Nn \fp_use:N { c }
17311 \cs_new_eq:NN \fp_eval:n \fp_to_decimal:n

(End definition for `\fp_use:N` and `\fp_eval:n`. These functions are documented on page 181.)

\fp_abs:n Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `__fp_parse:n`, namely, for better error reporting.

```

17312 \cs_new:Npn \fp_abs:n #1
17313 { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }

```

(End definition for `\fp_abs:n`. This function is documented on page 196.)

\fp_max:nn Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, etc.
\fp_min:nn 17314 \cs_new:Npn \fp_max:nn #1#2
17315 { \fp_to_decimal:n { max (__fp_parse:n {#1} , __fp_parse:n {#2}) } }
17316 \cs_new:Npn \fp_min:nn #1#2
17317 { \fp_to_decimal:n { min (__fp_parse:n {#1} , __fp_parse:n {#2}) } }

(End definition for `\fp_max:nn` and `\fp_min:nn`. These functions are documented on page 196.)

32.10 Convert an array of floating points to a comma list

`_fp_array_to_clist:n` Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```
\cs_new:Npn \_fp_array_to_clist_loop:Nw #1#2;
{
  \use_none:n #1
  { , ~ } \fp_to_tl:n { #1 #2 ; }
  \_fp_array_to_clist_loop:Nw
}
```

The `\use_ii:nn` function is expanded after `_fp_expand:n` is done, and it removes `,~` from the start of the representation.

```
17318 \cs_new:Npn \_fp_array_to_clist:n #1
17319 {
17320   \tl_if_empty:nF {#1}
17321   {
17322     \_fp_expand:n
17323     {
17324       { \use_ii:nn }
17325       \_fp_array_to_clist_loop:Nw #1 { ? \prg_break: } ;
17326       \prg_break_point:
17327     }
17328   }
17329 }
17330 \cs_new:Npx \_fp_array_to_clist_loop:Nw #1#2;
17331 {
17332   \exp_not:N \use_none:n #1
17333   \exp_not:N \exp_after:wN
17334   {
17335     \exp_not:N \exp_after:wN ,
17336     \exp_not:N \exp_after:wN \c_space_tl
17337     \exp_not:N \exp:w
17338     \exp_not:N \exp_end_continue_f:w
17339     \exp_not:N \_fp_to_tl_dispatch:w #1 #2 ;
17340   }
17341   \exp_not:N \_fp_array_to_clist_loop:Nw
17342 }
```

(End definition for `_fp_array_to_clist:n` and `_fp_array_to_clist_loop:Nw`.)

```
17343 </initex | package>
```

33 l3fp-random Implementation

```
17344 <*initex | package>
```

```
17345 <@@=fp>
```

`_fp_parse_word_rand:N` Those functions may receive a variable number of arguments. We won't use the argument `?`.

```
17346 \cs_new:Npn \_fp_parse_word_rand:N
17347 { \_fp_parse_function:NNN \_fp_rand_o:Nw ? }
17348 \cs_new:Npn \_fp_parse_word_randint:N
17349 { \_fp_parse_function:NNN \_fp_randint_o:Nw ? }
```

(End definition for `_fp_parse_word_rand:N` and `_fp_parse_word_randint:N`.)

33.1 Engine support

At present, X_YTeX does not provide random numbers, while pdfTeX, LuaTeX, pTeX and upTeX provide the primitive `\pdfTeX_uniformdeviate:D`. We write the test twice simply in order to write the false branch first.

```

17350 \cs_if_exist:NF \pdfTeX_uniformdeviate:D
17351 {
17352   \__kernel_msg_new:nnn { kernel } { fp-no-random }
17353   { Random~numbers~unavailable }
17354   \cs_new:Npn \_fp_rand_o:Nw ? #1 @
17355   {
17356     \__kernel_msg_expandable_error:nn { kernel } { fp-no-random }
17357     \exp_after:wN \c_nan_fp
17358   }
17359   \cs_new_eq:NN \_fp_randint_o:Nw \_fp_rand_o:Nw
17360   \cs_new:Npn \int_rand:nn #1#2
17361   {
17362     \__kernel_msg_expandable_error:nn { kernel } { fp-no-random }
17363     \int_eval:n {#1}
17364   }
17365 }
17366 \cs_if_exist:NT \pdfTeX_uniformdeviate:D
17367 {

```

`_fp_rand_uniform:`
`_fp_rand_size_int`
`_fp_rand_four_int`
`_fp_rand_eight_int`

The `\pdfTeX_uniformdeviate:D` primitive gives a pseudo-random integer in a range $[0, n - 1]$ of the user's choice. This number is meant to be uniformly distributed, but is produced by rescaling a uniform pseudo-random integer in $[0, 2^{28} - 1]$. For instance, setting n to (any multiple of) 2^{29} gives only even values. Thus it is only safe to call `\pdfTeX_uniformdeviate:D` with argument 2^{28} . This integer is also used in the implementation of `\int_rand:nn`. We also use variants of this number rounded down to multiples of 10^4 and 10^8 .

```

17368   \cs_new:Npn \_fp_rand_uniform:
17369   { \pdfTeX_uniformdeviate:D \c_fp_rand_size_int }
17370   \int_const:Nn \c_fp_rand_size_int { 268 435 456 }
17371   \int_const:Nn \c_fp_rand_four_int { 268 430 000 }
17372   \int_const:Nn \c_fp_rand_eight_int { 200 000 000 }

```

(End definition for `_fp_rand_uniform:` and others.)

`_fp_rand_myriads:n`
`_fp_rand_myriads_loop:nn`
`_fp_rand_myriads_get:w`
`_fp_rand_myriads_last:`
`_fp_rand_myriads_last:w`

Used as `_fp_rand_myriads:n {XXX}` with one input character per block of four digit we want. Given a pseudo-random integer from the primitive, we extract 2 blocks of digits if possible, namely if the integer is less than 2×10^8 . If that's not possible, we try to extract 1 block, which succeeds in the range $[2 \times 10^8, 26843 \times 10^4]$. For the 5456 remaining possible values we just throw away the random integer and get a new one. Depending on whether we got 2, 1, or 0 blocks, remove the same number of characters from the input stream with `\use_i:nnn`, `\use_i:nn` or nothing.

```

17373   \cs_new:Npn \_fp_rand_myriads:n #1
17374   {
17375     \_fp_rand_myriads_loop:nn #1
17376     { ? \use_i_delimit_by_q_stop:nw \_fp_rand_myriads_last: }

```

```

17377         { ? \use_none_delimit_by_q_stop:w } \q_stop
17378     }
17379 \cs_new:Npn \__fp_rand_myriads_loop:nn #1#2
17380 {
17381     \use_none:n #2
17382     \exp_after:wN \__fp_rand_myriads_get:w
17383     \int_value:w \__fp_rand_uniform: ; {#1}{#2}
17384 }
17385 \cs_new:Npn \__fp_rand_myriads_get:w #1 ;
17386 {
17387     \if_int_compare:w #1 < \c__fp_rand_eight_int
17388         \exp_after:wN \use_none:n
17389         \int_value:w \__fp_int_eval:w
17390         \c__fp_rand_eight_int + #1 \__fp_int_eval_end:
17391         \exp_after:wN \use_i:nnn
17392     \else:
17393         \if_int_compare:w #1 < \c__fp_rand_four_int
17394             \exp_after:wN \use_none:nnnnn
17395             \int_value:w \__fp_int_eval:w
17396             \c__fp_rand_four_int + #1 \__fp_int_eval_end:
17397             \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
17398         \fi:
17399     \fi:
17400     \__fp_rand_myriads_loop:nn
17401 }
17402 \cs_new:Npn \__fp_rand_myriads_last:
17403 {
17404     \exp_after:wN \__fp_rand_myriads_last:w
17405     \int_value:w \__fp_rand_uniform: ;
17406 }
17407 \cs_new:Npn \__fp_rand_myriads_last:w #1 ;
17408 {
17409     \if_int_compare:w #1 < \c__fp_rand_four_int
17410         \exp_after:wN \use_none:nnnnn
17411         \int_value:w \__fp_int_eval:w
17412         \c__fp_rand_four_int + #1 \__fp_int_eval_end:
17413     \else:
17414         \exp_after:wN \__fp_rand_myriads_last:
17415     \fi:
17416 }

```

(End definition for __fp_rand_myriads:n and others.)

33.2 Random floating point

__fp_rand_o:Nw First we check that random was called without argument. Then get four blocks of four digits.

```

\__fp_rand_o:w
17417 \cs_new:Npn \__fp_rand_o:Nw ? #1 @
17418 {
17419     \tl_if_empty:nTF {#1}
17420     { \__fp_rand_o: }
17421     {
17422         \__kernel_msg_expandable_error:nnnnn
17423         { kernel } { fp-num-args } { rand() } { 0 } { 0 }

```

```

17424         \exp_after:wN \c_nan_fp
17425     }
17426 }
17427 \cs_new:Npn \__fp_rand_o:
17428 { \__fp_parse_o:n { . \__fp_rand_myriads:n { xxxx } } }

```

(End definition for `__fp_rand_o:Nw`, `__fp_rand_o:`, and `__fp_rand_o:w`.)

33.3 Random integer

```

\__fp_randint_o:Nw
\__fp_randint_default:w
\__fp_randint_badarg:w
\__fp_randint_o:w
\__fp_randint_e:wnn
\__fp_randint_e:wwNnn
\__fp_randint_e:wwwNnn
\__fp_randint_narrow_e:nnnn
\__fp_randint_wide_e:nnnn
\__fp_randint_wide_e:wnnn

```

Enforce that there is one argument (then add first argument 1) or two arguments. Enforce that they are integers in $(-10^{16}, 10^{16})$ and ordered. We distinguish narrow ranges (less than 2^{28}) from wider ones.

For narrow ranges, compute the number n of possible outputs as an integer using `\fp_to_int:n`, and reduce a pseudo-random 28-bit integer r modulo n . On its own, this is not uniform when $[0, 2^{28} - 1]$ does not divide evenly into intervals of size n . The auxiliary `__fp_randint_e:wwwNnn` discards the pseudo-random integer if it lies in an incomplete interval, and repeats.

For wide ranges we use the same code except for the last eight digits which use `__fp_rand_myriads:n`. It is not safe to combine the first digits with the last eight as a single string of digits, as this may exceed 16 digits and be rounded. Instead, we first add the first few digits (times 10^8) to the lower bound. The result is compared to the upper bound and the process repeats if needed.

```

17429 \cs_new:Npn \__fp_randint_o:Nw ?
17430 {
17431     \__fp_parse_function_one_two:nnw
17432     { randint }
17433     { \__fp_randint_default:w \__fp_randint_o:w }
17434 }
17435 \cs_new:Npn \__fp_randint_default:w #1 { \exp_after:wN #1 \c_one_fp }
17436 \cs_new:Npn \__fp_randint_badarg:w \s__fp \__fp_chk:w #1#2#3;
17437 {
17438     \__fp_int:wTF \s__fp \__fp_chk:w #1#2#3;
17439     {
17440         \if_meaning:w 1 #1
17441         \if_int_compare:w
17442             \use_i_delimit_by_q_stop:nw #3 \q_stop > \c__fp_prec_int
17443             1 \exp_stop_f:
17444         \fi:
17445         \fi:
17446     }
17447     { 1 \exp_stop_f: }
17448 }
17449 \cs_new:Npn \__fp_randint_o:w #1; #2; @
17450 {
17451     \if_case:w
17452         \__fp_randint_badarg:w #1;
17453         \__fp_randint_badarg:w #2;
17454         \fp_compare:nNnTF { #1; } > { #2; } { 1 } { 0 } \exp_stop_f:
17455         \exp_after:wN \exp_after:wN \exp_after:wN \__fp_randint_e:wnn
17456         \__fp_parse:n { #2; - #1; } { #1; } { #2; }
17457     \or:
17458         \__fp_invalid_operation_tl_o:ff

```

```

17459         { randint } { \__fp_array_to_clist:n { #1; #2; } }
17460         \exp:w
17461         \fi:
17462         \exp_after:wN \exp_end:
17463     }
17464 \cs_new:Npn \__fp_randint_e:wnn #1;
17465 {
17466     \exp_after:wN \__fp_randint_e:wwNnn
17467     \int_value:w \__fp_rand_uniform: \exp_after:wN ;
17468     \exp:w \exp_end_continue_f:w
17469     \fp_compare:nNnTF { #1 ; } < \c__fp_rand_size_int
17470     { \fp_to_int:n { #1 ; + 1 } ; \__fp_randint_narrow_e:nnnn }
17471     {
17472         \fp_to_int:n { floor(#1 ; * 1e-8 + 1) } ;
17473         \__fp_randint_wide_e:nnnn
17474     }
17475 }
17476 \cs_new:Npn \__fp_randint_e:wwNnn #1 ; #2 ;
17477 {
17478     \exp_after:wN \__fp_randint_e:wwwNnn
17479     \int_value:w \int_mod:nn {#1} {#2} ; #1 ; #2 ;
17480 }
17481 \cs_new:Npn \__fp_randint_e:wwwNnn #1 ; #2 ; #3 ; #4
17482 {
17483     \int_compare:nNnTF { #2 - #1 + #3 } > \c__fp_rand_size_int
17484     {
17485         \exp_after:wN \__fp_randint_e:wwNnn
17486         \int_value:w \__fp_rand_uniform: ; #3 ; #4
17487     }
17488     { #4 {#1} {#3} }
17489 }
17490 \cs_new:Npn \__fp_randint_narrow_e:nnnn #1#2#3#4
17491 { \__fp_parse_o:n { #3 + #1 } \exp:w }
17492 \cs_new:Npn \__fp_randint_wide_e:nnnn #1#2#3#4
17493 {
17494     \exp_after:wN \exp_after:wN
17495     \exp_after:wN \__fp_randint_wide_e:wnnn
17496     \__fp_parse:n { #3 + #1e8 + \__fp_rand_myriads:n { xx } }
17497     {#2} {#3} {#4}
17498 }
17499 \cs_new:Npn \__fp_randint_wide_e:wnnn #1 ; #2#3#4
17500 {
17501     \fp_compare:nNnTF { #1 ; } > {#4}
17502     {
17503         \exp_after:wN \__fp_randint_e:wwNnn
17504         \int_value:w \__fp_rand_uniform: ; #2 ;
17505         \__fp_randint_wide_e:nnnn {#3} {#4}
17506     }
17507     { \__fp_exp_after_o:w #1 ; \exp:w }
17508 }

```

(End definition for __fp_randint_o:Nw and others.)

\int_rand:nn Evaluate the argument and filter out the case where the lower bound #1 is more than the upper bound #2. Then determine whether the range is narrower than \c__fp_rand_
__fp_randint:ww
__fp_randint_narrow:n
__fp_randint_narrow:nn
__fp_randint_narrow:nnn

`size_int`; `#2-#1` may overflow for very large positive `#2` and negative `#1`. If the range is wide, use slower code from `l3fp`. If the range is narrow, call `__fp_randint_narrow:n` $\{ \langle \textit{choices} \rangle \}$ where $\langle \textit{choices} \rangle$ is the number of possible outcomes.

One option then would be to reduce a random number modulo $\langle \textit{choices} \rangle$, but the low-order bits of random numbers provided by the primitive have poor randomness: empirical tests indicate that taking random numbers modulo 2 gives only two possible sequences of bits depending on the seed's parity. Instead, fit in the output range $[0, \text{c___fp_rand_size_int} - 1]$ of the random number generator $\langle \textit{choices} \rangle$ equal-size intervals of maximum possible size (obtained using `\int_div_truncate:nn`). Given a random number, divide it by the size to determine which interval it falls into; if it is beyond $\langle \textit{choices} \rangle$, try another random number.

```

17509 \cs_new:Npn \int_rand:nn #1#2
17510 {
17511   \exp_after:wN \__fp_randint:ww
17512   \int_value:w \int_eval:n {#1} \exp_after:wN ;
17513   \int_value:w \int_eval:n {#2} ;
17514 }
17515 \cs_new:Npn \__fp_randint:ww #1; #2;
17516 {
17517   \int_compare:nNnTF {#1} > {#2}
17518   {
17519     \__kernel_msg_expandable_error:nnnn
17520     { kernel } { randint-backward-range } {#1} {#2}
17521     \__fp_randint:ww #2; #1;
17522   }
17523   {
17524     \int_compare:nNnTF {#1} > 0
17525     { \int_compare:nNnTF { #2 - #1 } < \c__fp_rand_size_int }
17526     { \int_compare:nNnTF {#2} < { #1 + \c__fp_rand_size_int } }
17527     {
17528       \int_eval:n
17529       {
17530         #1 +
17531         \exp_args:Nf \__fp_randint_narrow:n
17532         { \int_eval:n { #2 - #1 + 1 } }
17533       }
17534     }
17535     { \fp_to_int:n { randint(#1,#2) } }
17536   }
17537 }
17538 \cs_new:Npn \__fp_randint_narrow:n #1
17539 {
17540   \exp_args:Nf \__fp_randint_narrow:nn
17541   { \int_div_truncate:nn \c__fp_rand_size_int {#1} } {#1}
17542 }
17543 \cs_new:Npn \__fp_randint_narrow:nn #1
17544 {
17545   \exp_args:Nf \__fp_randint_narrow:nnn
17546   {
17547     \int_div_truncate:nn
17548     { \pdfTeX_uniformdeviate:D \c__fp_rand_size_int } {#1}
17549   }
17550   {#1}

```

```

17551     }
17552     \cs_new:Npn \__fp_randint_narrow:nnn #1#2#3
17553     {
17554         \int_compare:nNnTF {#1} < {#3}
17555         {#1}
17556         { \__fp_randint_narrow:nn {#2} {#3} }
17557     }

```

(End definition for `\int_rand:nn` and others. This function is documented on page 86.)

End the initial conditional that ensures these commands are only defined in engines that support random numbers.

```

17558     }
17559 \</initex | package>

```

34 l3intarray implementation

```

17560 \*initex | package>
17561 \@@=intarray>

```

34.1 Allocating arrays

`\c__intarray_sp_dim` Used to convert integers to dimensions fast.

```

17562 \dim_const:Nn \c__intarray_sp_dim { 1 sp }

```

(End definition for `\c__intarray_sp_dim`.)

`\g__intarray_font_int` Used to assign one font per array.

```

17563 \int_new:N \g__intarray_font_int

```

(End definition for `\g__intarray_font_int`.)

`\intarray_new:Nn` Declare `#1` to be a font (arbitrarily `cmr10` at a never-used size). Store the array's size as the `\hyphenchar` of that font and make sure enough `\fontdimen` are allocated, by setting the last one. Then clear any `\fontdimen` that `cmr10` starts with. It seems LuaTeX's `cmr10` has an extra `\fontdimen` parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such). Every `intarray` must be global; it's enough to run this check in `\intarray_new:Nn`.

```

17564 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN g #1 } { }
17565 \cs_new_protected:Npn \intarray_new:Nn #1#2
17566 {
17567     \__kernel_chk_if_free_cs:N #1
17568     \int_gincr:N \g__intarray_font_int
17569     \tex_global:D \tex_font:D #1
17570     = cmr10~at~ \g__intarray_font_int \c__intarray_sp_dim \scan_stop:
17571     \tex_hyphenchar:D #1 = \int_eval:n {#2} \scan_stop:
17572     \int_compare:nNnT { \tex_hyphenchar:D #1 } > 0
17573     { \tex_fontdimen:D \tex_hyphenchar:D #1 #1 = \c_zero_dim }
17574     \int_step_inline:nn { 8 } { \tex_fontdimen:D ##1 #1 = \c_zero_dim }
17575 }

```

(End definition for `\intarray_new:Nn`. This function is documented on page 199.)

`\intarray_count:N` Size of an array.

```

17576 \cs_new:Npn \intarray_count:N #1 { \tex_the:D \tex_hyphenchar:D #1 }

```

(End definition for `\intarray_count:N`. This function is documented on page 199.)

34.2 Array items

The functions `\intarray_gset:Nnn` and `\intarray_item:Nn` share bounds checking. The T branch is used if #3 is within bounds of the array #2.

```

17577 \cs_new:Npn \__intarray_bounds:NNnTF #1#2#3#4#5
17578 {
17579   \if_int_compare:w 1 > #3 \exp_stop_f:
17580     \__intarray_bounds_error:NNn #1 #2 {#3}
17581     #5
17582   \else:
17583     \if_int_compare:w #3 > \intarray_count:N #2 \exp_stop_f:
17584     \__intarray_bounds_error:NNn #1 #2 {#3}
17585     #5
17586   \else:
17587     #4
17588   \fi:
17589 \fi:
17590 }
17591 \cs_new:Npn \__intarray_bounds_error:NNn #1#2#3
17592 {
17593   #1 { kernel } { out-of-bounds }
17594   { \token_to_str:N #2 } {#3} { \intarray_count:N #2 }
17595 }
```

(End definition for `__intarray_bounds:NNnTF` and `__intarray_bounds_error:NNn`.)

Set the appropriate `\fontdimen`. The `__kernel_intarray_gset:Nnn` function does not use `\int_eval:n`, namely its arguments must be suitable for `\int_value:w`. The user version checks the position and value are within bounds.

```

17596 \cs_new_protected:Npn \__kernel_intarray_gset:Nnn #1#2#3
17597 { \tex_fontdimen:D #2 #1 #3 \c__intarray_sp_dim }
17598 \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
17599 {
17600   \exp_after:wN \__intarray_gset:Nww
17601   \exp_after:wN #1
17602   \int_value:w \int_eval:n {#2} \exp_after:wN ;
17603   \int_value:w \int_eval:n {#3} ;
17604 }
17605 \cs_new_protected:Npn \__intarray_gset:Nww #1#2 ; #3 ;
17606 {
17607   \__intarray_bounds:NNnTF \__kernel_msg_error:nnxxx #1 {#2}
17608   {
17609     \if_int_compare:w \int_abs:n {#3} > \c_max_dim
17610     \exp_after:wN \__intarray_gset_overflow:NNnn
17611     \fi:
17612     \__kernel_intarray_gset:Nnn #1 {#2} {#3}
17613   }
17614   { }
17615 }
17616 \cs_new_protected:Npn \__intarray_gset_overflow:NNnn #1#2#3#4
17617 {
17618   \__kernel_msg_error:nnxxxx { kernel } { overflow }
17619   { \token_to_str:N #2 } {#3} {#4}
17620   { \int_value:w \int_compare:nNnT {#4} < 0 { - } \c_max_dim }
```

```

17621 \__kernel_intarray_gset:Nnn #2 {#3}
17622 { \int_value:w \int_compare:nNnT {#4} < 0 { - } \c_max_dim }
17623 }

```

(End definition for `\intarray_gset:Nnn` and others. This function is documented on page 199.)

`\intarray_item:Nn` Get the appropriate `\fontdimen` and perform bound checks. The `__kernel_intarray_item:Nn` function omits bound checks and omits `\int_eval:n`, namely its argument must be a TeX integer suitable for `\int_value:w`.

```

17624 \cs_new:Npn \__kernel_intarray_item:Nn #1#2
17625 { \int_value:w \tex_fontdimen:D #2 #1 }
17626 \cs_new:Npn \intarray_item:Nn #1#2
17627 {
17628   \exp_after:wN \__intarray_item:Nw
17629   \exp_after:wN #1
17630   \int_value:w \int_eval:n {#2} ;
17631 }
17632 \cs_new:Npn \__intarray_item:Nw #1#2 ;
17633 {
17634   \__intarray_bounds:NNnTF \__kernel_msg_expandable_error:nnfff #1 {#2}
17635   { \__kernel_intarray_item:Nn #1 {#2} }
17636   { 0 }
17637 }

```

(End definition for `\intarray_item:Nn`, `__kernel_intarray_item:Nn`, and `__intarray_item:Nn`. This function is documented on page 199.)

```

17638 </initex | package>

```

35 l3sort implementation

```

17639 <*initex | package>
17640 <@@=sort>

```

35.1 Variables

`\g__sort_internal_seq` `\g__sort_internal_tl` Sorting happens in a group; the result is stored in those global variables before being copied outside the group to the proper places. For `seq` and `tl` this is more efficient than using `\use:x` (or some `\exp_args:NNNx`) to smuggle the definition outside the group since TeX does not need to re-read tokens. For `clist` we don't gain anything since the result is converted from `seq` to `clist` anyways.

```

17641 \seq_new:N \g__sort_internal_seq
17642 \tl_new:N \g__sort_internal_tl

```

(End definition for `\g__sort_internal_seq` and `\g__sort_internal_tl`.)

`\l__sort_length_int` `\l__sort_min_int` `\l__sort_top_int` `\l__sort_max_int` `\l__sort_true_max_int` The sequence has `\l__sort_length_int` items and is stored from `\l__sort_min_int` to `\l__sort_top_int - 1`. While reading the sequence in memory, we check that `\l__sort_top_int` remains at most `\l__sort_max_int`, precomputed by `__sort_compute_range:.` That bound is such that the merge sort only uses `\toks` registers less than `\l__sort_true_max_int`, namely those that have not been allocated for use in other code: the user's comparison code could alter these.

```

17643 \int_new:N \l__sort_length_int
17644 \int_new:N \l__sort_min_int

```

```

17645 \int_new:N \l__sort_top_int
17646 \int_new:N \l__sort_max_int
17647 \int_new:N \l__sort_true_max_int

```

(End definition for `\l__sort_length_int` and others.)

`\l__sort_block_int` Merge sort is done in several passes. In each pass, blocks of size `\l__sort_block_int` are merged in pairs. The block size starts at 1, and, for a length in the range $[2^k + 1, 2^{k+1}]$, reaches 2^k in the last pass.

```

17648 \int_new:N \l__sort_block_int

```

(End definition for `\l__sort_block_int`.)

`\l__sort_begin_int` `\l__sort_end_int` When merging two blocks, `\l__sort_begin_int` marks the lowest index in the two blocks, and `\l__sort_end_int` marks the highest index, plus 1.

```

17649 \int_new:N \l__sort_begin_int
17650 \int_new:N \l__sort_end_int

```

(End definition for `\l__sort_begin_int` and `\l__sort_end_int`.)

`\l__sort_A_int` `\l__sort_B_int` `\l__sort_C_int` When merging two blocks (whose end-points are `beg` and `end`), *A* starts from the high end of the low block, and decreases until reaching `beg`. The index *B* starts from the top of the range and marks the register in which a sorted item should be put. Finally, *C* points to the copy of the high block in the interval of registers starting at `\l__sort_length_int`, upwards. *C* starts from the upper limit of that range.

```

17651 \int_new:N \l__sort_A_int
17652 \int_new:N \l__sort_B_int
17653 \int_new:N \l__sort_C_int

```

(End definition for `\l__sort_A_int`, `\l__sort_B_int`, and `\l__sort_C_int`.)

35.2 Finding available `\toks` registers

`__sort_shrink_range:` `__sort_shrink_range_loop:` After `__sort_compute_range:` (defined below) determines that `\toks` registers between `\l__sort_min_int` (included) and `\l__sort_true_max_int` (excluded) have not yet been assigned, `__sort_shrink_range:` computes `\l__sort_max_int` to reflect the need for a buffer when merging blocks in the merge sort. Given $2^n \leq A \leq 2^n + 2^{n-1}$ registers we can sort $\lfloor A/2 \rfloor + 2^{n-2}$ items while if we have $2^n + 2^{n-1} \leq A \leq 2^{n+1}$ registers we can sort $A - 2^{n-1}$ items. We first find out a power 2^n such that $2^n \leq A \leq 2^{n+1}$ by repeatedly halving `\l__sort_block_int`, starting at 2^{15} or 2^{14} namely half the total number of registers, then we use the formulas and set `\l__sort_max_int`.

```

17654 \cs_new_protected:Npn \__sort_shrink_range:
17655 {
17656   \int_set:Nn \l__sort_A_int
17657     { \l__sort_true_max_int - \l__sort_min_int + 1 }
17658   \int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
17659   \__sort_shrink_range_loop:
17660   \int_set:Nn \l__sort_max_int
17661     {
17662       \int_compare:nNnTF
17663         { \l__sort_block_int * 3 / 2 } > \l__sort_A_int
17664         {
17665           \l__sort_min_int

```

```

17666         + ( \l__sort_A_int - 1 ) / 2
17667         + \l__sort_block_int / 4
17668         - 1
17669     }
17670     { \l__sort_true_max_int - \l__sort_block_int / 2 }
17671 }
17672 }
17673 \cs_new_protected:Npn \__sort_shrink_range_loop:
17674 {
17675     \if_int_compare:w \l__sort_A_int < \l__sort_block_int
17676         \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
17677         \exp_after:wN \__sort_shrink_range_loop:
17678     \fi:
17679 }

```

(End definition for __sort_shrink_range: and __sort_shrink_range_loop:.)

__sort_compute_range: First find out what \toks have not yet been assigned. There are many cases. In L^AT_EX 2_ε with no package, available \toks range from \count15 + 1 to \c_max_register_int included (this was not altered despite the 2015 changes). When \loctoks is defined, namely in plain (e)T_EX, or when the package etex is loaded in L^AT_EX 2_ε, redefine __sort_compute_range: to use the range \count265 to \count275 - 1. The elocalloc package also defines \loctoks but uses yet another number for the upper bound, namely \e@alloc@top (minus one). We must check for \loctoks every time a sorting function is called, as etex or elocalloc could be loaded.

In ConT_EXt MkIV the range is from \c_syst_last_allocated_toks+1 to \c_max_register_int, and in MkII it is from \lastallocatedtoks+1 to \c_max_register_int. In all these cases, call __sort_shrink_range:. The L^AT_EX3 format mode is easiest: no \toks are ever allocated so available \toks range from 0 to \c_max_register_int and we precompute the result of __sort_shrink_range:.

```

17680 (*package)
17681 \cs_new_protected:Npn \__sort_compute_range:
17682 {
17683     \int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 }
17684     \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
17685     \__sort_shrink_range:
17686     \if_meaning:w \loctoks \tex_undefined:D \else:
17687         \if_meaning:w \loctoks \scan_stop: \else:
17688             \__sort_redefine_compute_range:
17689             \__sort_compute_range:
17690         \fi:
17691     \fi:
17692 }
17693 \cs_new_protected:Npn \__sort_redefine_compute_range:
17694 {
17695     \cs_if_exist:cTF { ver@elocalloc.sty }
17696     {
17697         \cs_gset_protected:Npn \__sort_compute_range:
17698         {
17699             \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
17700             \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
17701             \__sort_shrink_range:
17702         }

```

```

17703     }
17704     {
17705         \cs_gset_protected:Npn \__sort_compute_range:
17706         {
17707             \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
17708             \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
17709             \__sort_shrink_range:
17710         }
17711     }
17712 }
17713 \cs_if_exist:NT \loctoks { \__sort_redefine_compute_range: }
17714 \tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks }
17715 {
17716     \cs_if_exist:NT #1
17717     {
17718         \cs_gset_protected:Npn \__sort_compute_range:
17719         {
17720             \int_set:Nn \l__sort_min_int { #1 + 1 }
17721             \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
17722             \__sort_shrink_range:
17723         }
17724     }
17725 }
17726 \end{package}
17727 \begin{initex}
17728 \int_const:Nn \c__sort_max_length_int
17729 { ( \c_max_register_int + 1 ) * 3 / 4 }
17730 \cs_new_protected:Npn \__sort_compute_range:
17731 {
17732     \int_set:Nn \l__sort_min_int { 0 }
17733     \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
17734     \int_set:Nn \l__sort_max_int { \c__sort_max_length_int }
17735 }
17736 \end{initex}

```

(End definition for `__sort_compute_range:`, `__sort_redefine_compute_range:`, and `\c__sort_max_length_int`.)

35.3 Protected user commands

`__sort_main:NNNn` Sorting happens in three steps. First store items in `\toks` registers ranging from `\l__sort_min_int` to `\l__sort_top_int - 1`, while checking that the list is not too long. If we reach the maximum length, that's an error; exit the group. Secondly, sort the array of `\toks` registers, using the user-defined sorting function: `__sort_level:` calls `__sort_compare:nn` as needed. Finally, unpack the `\toks` registers (now sorted) into the target `tl`, or into `\g__sort_internal_seq` for `seq` and `clist`. This is done by `__sort_seq:NNNNn` and `__sort_tl:NNn`.

```

17737 \cs_new_protected:Npn \__sort_main:NNNn #1#2#3#4
17738 {
17739     \package \__sort_disable_tokstokdef:
17740     \__sort_compute_range:
17741     \int_set_eq:NN \l__sort_top_int \l__sort_min_int
17742     #1 #3
17743     {

```

```

17744         \if_int_compare:w \l__sort_top_int = \l__sort_max_int
17745         \__sort_too_long_error:NNw #2 #3
17746         \fi:
17747         \tex_toks:D \l__sort_top_int {##1}
17748         \int_incr:N \l__sort_top_int
17749     }
17750     \int_set:Nn \l__sort_length_int
17751     { \l__sort_top_int - \l__sort_min_int }
17752     \cs_set:Npn \__sort_compare:nn ##1 ##2 {#4}
17753     \int_set:Nn \l__sort_block_int { 1 }
17754     \__sort_level:
17755 }

```

(End definition for __sort_main:NNNn.)

\tl_sort:Nn Call the main sorting function then unpack \toks registers outside the group into the target token list. The unpacking is done by __sort_tl_toks:w; registers are numbered from \l__sort_min_int to \l__sort_top_int - 1. For expansion behaviour we need a couple of primitives. The \tl_gclear:N reduces memory usage. The \prg_break_point: is used by __sort_main:NNNn when the list is too long.

```

\__sort_tl:NNn
\__sort_tl_toks:w
17756 \cs_new_protected:Npn \tl_sort:Nn { \__sort_tl:NNn \tl_set_eq:NN }
17757 \cs_generate_variant:Nn \tl_sort:Nn { c }
17758 \cs_new_protected:Npn \tl_gsort:Nn { \__sort_tl:NNn \tl_gset_eq:NN }
17759 \cs_generate_variant:Nn \tl_gsort:Nn { c }
17760 \cs_new_protected:Npn \__sort_tl:NNn #1#2#3
17761 {
17762     \group_begin:
17763     \__sort_main:NNNn \tl_map_inline:Nn \tl_map_break:n #2 {#3}
17764     \tl_gset:Nx \g__sort_internal_tl
17765     { \__sort_tl_toks:w \l__sort_min_int ; }
17766     \group_end:
17767     #1 #2 \g__sort_internal_tl
17768     \tl_gclear:N \g__sort_internal_tl
17769     \prg_break_point:
17770 }
17771 \cs_new:Npn \__sort_tl_toks:w #1 ;
17772 {
17773     \if_int_compare:w #1 < \l__sort_top_int
17774     { \tex_the:D \tex_toks:D #1 }
17775     \exp_after:wN \__sort_tl_toks:w
17776     \int_value:w \int_eval:n { #1 + 1 } \exp_after:wN ;
17777     \fi:
17778 }

```

(End definition for \tl_sort:Nn and others. These functions are documented on page 43.)

\seq_sort:Nn Use the same general framework for seq and clist. Apply the general sorting code, then unpack \toks into \g__sort_internal_seq. Outside the group copy or convert (for clist) the data to the target variable. The \seq_gclear:N reduces memory usage. The \prg_break_point: is used by __sort_main:NNNn when the list is too long.

```

\seq_gsort:Nn
\clist_sort:Nn
\clist_gsort:Nn
17779 \cs_new_protected:Npn \seq_sort:Nn
17780 { \__sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_set_eq:NN }
17781 \cs_generate_variant:Nn \seq_sort:Nn { c }
17782 \cs_new_protected:Npn \seq_gsort:Nn
\__sort_seq:NNNNn

```

```

17783 { \_sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_gset_eq:NN }
17784 \cs_generate_variant:Nn \seq_gsort:Nn { c }
17785 \cs_new_protected:Npn \clist_sort:Nn
17786 {
17787   \_sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
17788   \clist_set_from_seq:NN
17789 }
17790 \cs_generate_variant:Nn \clist_sort:Nn { c }
17791 \cs_new_protected:Npn \clist_gsort:Nn
17792 {
17793   \_sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
17794   \clist_gset_from_seq:NN
17795 }
17796 \cs_generate_variant:Nn \clist_gsort:Nn { c }
17797 \cs_new_protected:Npn \_sort_seq:NNNNn #1#2#3#4#5
17798 {
17799   \group_begin:
17800   \_sort_main:NNNn #1 #2 #4 {#5}
17801   \seq_gset_from_inline_x:Nnn \g__sort_internal_seq
17802   {
17803     \int_step_function:nnN
17804     { \l__sort_min_int } { \l__sort_top_int - 1 }
17805   }
17806   { \tex_the:D \tex_toks:D ##1 }
17807   \group_end:
17808   #3 #4 \g__sort_internal_seq
17809   \seq_gclear:N \g__sort_internal_seq
17810   \prg_break_point:
17811 }

```

(End definition for `\seq_sort:Nn` and others. These functions are documented on page 69.)

35.4 Merge sort

`_sort_level:` This function is called once blocks of size `\l__sort_block_int` (initially 1) are each sorted. If the whole list fits in one block, then we are done (this also takes care of the case of an empty list or a list with one item). Otherwise, go through pairs of blocks starting from 0, then double the block size, and repeat.

```

17812 \cs_new_protected:Npn \_sort_level:
17813 {
17814   \if_int_compare:w \l__sort_block_int < \l__sort_length_int
17815     \l__sort_end_int \l__sort_min_int
17816     \_sort_merge_blocks:
17817     \tex_advance:D \l__sort_block_int \l__sort_block_int
17818     \exp_after:wN \_sort_level:
17819   \fi:
17820 }

```

(End definition for `_sort_level:.`)

`_sort_merge_blocks:` This function is called to merge a pair of blocks, starting at the last value of `\l__sort_end_int` (end-point of the previous pair of blocks). If shifting by one block to the right we reach the end of the list, then this pass has ended: the end of the list is sorted already. Otherwise, store the result of that shift in *A*, which indexes the first

block starting from the top end. Then locate the end-point (maximum) of the second block: shift `end` upwards by one more block, but keeping it $\leq \text{top}$. Copy this upper block of `\toks` registers in registers above `length`, indexed by `C`: this is covered by `__sort_copy_block:`. Once this is done we are ready to do the actual merger using `__sort_merge_blocks_aux:`, after shifting `A`, `B` and `C` so that they point to the largest index in their respective ranges rather than pointing just beyond those ranges. Of course, once that pair of blocks is merged, move on to the next pair.

```

17821 \cs_new_protected:Npn \__sort_merge_blocks:
17822 {
17823   \l__sort_begin_int \l__sort_end_int
17824   \tex_advance:D \l__sort_end_int \l__sort_block_int
17825   \if_int_compare:w \l__sort_end_int < \l__sort_top_int
17826     \l__sort_A_int \l__sort_end_int
17827     \tex_advance:D \l__sort_end_int \l__sort_block_int
17828     \if_int_compare:w \l__sort_end_int > \l__sort_top_int
17829       \l__sort_end_int \l__sort_top_int
17830   \fi:
17831   \l__sort_B_int \l__sort_A_int
17832   \l__sort_C_int \l__sort_top_int
17833   \__sort_copy_block:
17834   \int_decr:N \l__sort_A_int
17835   \int_decr:N \l__sort_B_int
17836   \int_decr:N \l__sort_C_int
17837   \exp_after:wN \__sort_merge_blocks_aux:
17838   \exp_after:wN \__sort_merge_blocks:
17839   \fi:
17840 }

```

(End definition for `__sort_merge_blocks:`)

`__sort_copy_block:` We wish to store a copy of the “upper” block of `\toks` registers, ranging between the initial value of `\l__sort_B_int` (included) and `\l__sort_end_int` (excluded) into a new range starting at the initial value of `\l__sort_C_int`, namely `\l__sort_top_int`.

```

17841 \cs_new_protected:Npn \__sort_copy_block:
17842 {
17843   \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
17844   \int_incr:N \l__sort_C_int
17845   \int_incr:N \l__sort_B_int
17846   \if_int_compare:w \l__sort_B_int = \l__sort_end_int
17847     \use_i:nn
17848   \fi:
17849   \__sort_copy_block:
17850 }

```

(End definition for `__sort_copy_block:`)

`__sort_merge_blocks_aux:` At this stage, the first block starts at `\l__sort_begin_int`, and ends at `\l__sort_A_int`, and the second block starts at `\l__sort_top_int` and ends at `\l__sort_C_int`. The result of the merger is stored at positions indexed by `\l__sort_B_int`, which starts at `\l__sort_end_int - 1` and decreases down to `\l__sort_begin_int`, covering the full range of the two blocks. In other words, we are building the merger starting with the largest values. The comparison function is defined to return either `swapped` or `same`. Of

course, this means the arguments need to be given in the order they appear originally in the list.

```

17851 \cs_new_protected:Npn \__sort_merge_blocks_aux:
17852 {
17853   \exp_after:wN \__sort_compare:nn \exp_after:wN
17854   { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
17855   \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
17856   \prg_do_nothing:
17857   \__sort_return_mark:N
17858   \__sort_return_mark:N
17859   \__sort_return_none_error:
17860 }

```

(End definition for __sort_merge_blocks_aux:.)

```

\sort_return_same: The marker removes one token. Each comparison should call \sort_return_same: or
\sort_return_swapped: \sort_return_swapped: exactly once. If neither is called, \__sort_return_none_
\__sort_return_mark:N error: is called.
\__sort_return_none_error:
\__sort_return_two_error:w
17861 \cs_new_protected:Npn \sort_return_same: #1 \__sort_return_mark:N
17862 {
17863   #1
17864   \__sort_return_mark:N
17865   \__sort_return_two_error:w \__sort_return_same:
17866 }
17867 \cs_new_protected:Npn \sort_return_swapped: #1 \__sort_return_mark:N
17868 {
17869   #1
17870   \__sort_return_mark:N
17871   \__sort_return_two_error:w \__sort_return_swapped:
17872 }
17873 \cs_new_protected:Npn \__sort_return_mark:N #1 { }
17874 \cs_new_protected:Npn \__sort_return_none_error:
17875 {
17876   \__kernel_msg_error:nnxx { kernel } { return-none }
17877   { \tex_the:D \tex_toks:D \l__sort_A_int }
17878   { \tex_the:D \tex_toks:D \l__sort_C_int }
17879   \__sort_return_same:
17880 }
17881 \cs_new_protected:Npn \__sort_return_two_error:w
17882 #1 \__sort_return_none_error:
17883 { \__kernel_msg_error:nn { kernel } { return-two } }

```

(End definition for \sort_return_same: and others. These functions are documented on page 200.)

__sort_return_same: If the comparison function returns **same**, then the second argument fed to __sort_compare:nn should remain to the right of the other one. Since we build the merger starting from the right, we copy that \toks register into the allotted range, then shift the pointers *B* and *C*, and go on to do one more step in the merger, unless the second block has been exhausted: then the remainder of the first block is already in the correct registers and we are done with merging those two blocks.

```

17884 \cs_new_protected:Npn \__sort_return_same:
17885 {
17886   \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
17887   \int_decr:N \l__sort_B_int

```

```

17888 \int_decr:N \l__sort_C_int
17889 \if_int_compare:w \l__sort_C_int < \l__sort_top_int
17890 \use_i:nn
17891 \fi:
17892 \__sort_merge_blocks_aux:
17893 }

```

(End definition for __sort_return_same:.)

__sort_return_swapped: If the comparison function returns **swapped**, then the next item to add to the merger is the first argument, contents of the \toks register *A*. Then shift the pointers *A* and *B* to the left, and go for one more step for the merger, unless the left block was exhausted (*A* goes below the threshold). In that case, all remaining \toks registers in the second block, indexed by *C*, are copied to the merger by __sort_merge_blocks_end:.

```

17894 \cs_new_protected:Npn \__sort_return_swapped:
17895 {
17896 \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
17897 \int_decr:N \l__sort_B_int
17898 \int_decr:N \l__sort_A_int
17899 \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
17900 \__sort_merge_blocks_end: \use_i:nn
17901 \fi:
17902 \__sort_merge_blocks_aux:
17903 }

```

(End definition for __sort_return_swapped:.)

__sort_merge_blocks_end: This function's task is to copy the \toks registers in the block indexed by *C* to the merger indexed by *B*. The end can equally be detected by checking when *B* reaches the threshold **begin**, or when *C* reaches **top**.

```

17904 \cs_new_protected:Npn \__sort_merge_blocks_end:
17905 {
17906 \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
17907 \int_decr:N \l__sort_B_int
17908 \int_decr:N \l__sort_C_int
17909 \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
17910 \use_i:nn
17911 \fi:
17912 \__sort_merge_blocks_end:
17913 }

```

(End definition for __sort_merge_blocks_end:.)

35.5 Expandable sorting

Sorting expandably is very different from sorting and assigning to a variable. Since tokens cannot be stored, they must remain in the input stream, and be read through at every step. It is thus necessarily much slower (at best $O(n^2 \ln n)$) than non-expandable sorting functions ($O(n \ln n)$).

A prototypical version of expandable quicksort is as follows. If the argument has no item, return nothing, otherwise partition, using the first item as a pivot (argument #4 of __sort:nnNnn). The arguments of __sort:nnNnn are 1. items less than #4, 2. items greater or equal to #4, 3. comparison, 4. pivot, 5. next item to test. If #5 is the tail of

the list, call `\tl_sort:nN` on `#1` and on `#2`, placing `#4` in between; `\use:ff` expands the parts to make `\tl_sort:nN` f-expandable. Otherwise, compare `#4` and `#5` using `#3`. If they are ordered, place `#5` amongst the “greater” items, otherwise amongst the “lesser” items, and continue partitioning.

```
\cs_new:Npn \tl_sort:nN #1#2
{
  \tl_if_blank:nF {#1}
  {
    \__sort:nnNnn { } { } #2
    #1 \q_recursion_tail \q_recursion_stop
  }
}
\cs_new:Npn \__sort:nnNnn #1#2#3#4#5
{
  \quark_if_recursion_tail_stop_do:nn {#5}
  { \use:ff { \tl_sort:nN {#1} #3 {#4} } { \tl_sort:nN {#2} #3 } }
  #3 {#4} {#5}
  { \__sort:nnNnn {#1} { #2 {#5} } #3 {#4} }
  { \__sort:nnNnn { #1 {#5} } {#2} #3 {#4} }
}
\cs_generate_variant:Nn \use:nn { ff }
```

There are quite a few optimizations available here: the code below is less legible, but more than twice as fast.

In the simple version of the code, `__sort:nnNnn` is called $O(n \ln n)$ times on average (the number of comparisons required by the quicksort algorithm). Hence most of our focus is on optimizing that function.

The first speed up is to avoid testing for the end of the list at every call to `__sort:nnNnn`. For this, the list is prepared by changing each $\langle item \rangle$ of the original token list into $\langle command \rangle \{ \langle item \rangle \}$, just like sequences are stored. We arrange things such that the $\langle command \rangle$ is the $\langle conditional \rangle$ provided by the user: the loop over the $\langle prepared tokens \rangle$ then looks like

```
\cs_new:Npn \__sort_loop:wNn ... #6#7
{
  #6 { \langle pivot \rangle } {#7} \langle loop big \rangle \langle loop small \rangle
  \langle extra arguments \rangle
}
\__sort_loop:wNn ... \langle prepared tokens \rangle
\langle end-loop \rangle {} \q_stop
```

In this example, which matches the structure of `__sort_quick_split_i:NnnnnNn` and a few other functions below, the `__sort_loop:wNn` auxiliary normally receives the user’s $\langle conditional \rangle$ as `#6` and an $\langle item \rangle$ as `#7`. This is compared to the $\langle pivot \rangle$ (the argument `#5`, not shown here), and the $\langle conditional \rangle$ leaves the $\langle loop big \rangle$ or $\langle loop small \rangle$ auxiliary, which both have the same form as `__sort_loop:wNn`, receiving the next pair $\langle conditional \rangle \{ \langle item \rangle \}$ as `#6` and `#7`. At the end, `#6` is the $\langle end-loop \rangle$ function, which terminates the loop.

The second speed up is to minimize the duplicated tokens between the `true` and `false` branches of the conditional. For this, we introduce two versions of `__sort:nnNnn`,

which receive the new item as #1 and place it either into the list #2 of items less than the pivot #4 or into the list #3 of items greater or equal to the pivot.

```
\cs_new:Npn \__sort_i:nnnnNn #1#2#3#4#5#6
{
  #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
  {#6} { #2 {#1} } {#3} {#4}
}
\cs_new:Npn \__sort_ii:nnnnNn #1#2#3#4#5#6
{
  #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
  {#6} {#2} { #3 {#1} } {#4}
}
```

Note that the two functions have the form of `__sort_loop:wNn` above, receiving as #5 the conditional or a function to end the loop. In fact, the lists #2 and #3 must be made of pairs $\langle \text{conditional} \rangle \{ \langle \text{item} \rangle \}$, so we have to replace {#6} above by { #5 {#6} }, and {#1} by #1. The actual functions have one more argument, so all argument numbers are shifted compared to this code.

The third speed up is to avoid `\use:ff` using a continuation-passing style: `__sort_quick_split:NnNn` expects a list followed by `\q_mark {<code>}`, and expands to $\langle \text{code} \rangle \langle \text{sorted list} \rangle$. Sorting the two parts of the list around the pivot is done with

```
\__sort_quick_split:NnNn #2 ... \q_mark
{
  \__sort_quick_split:NnNn #1 ... \q_mark {<code>}
  {<pivot>}
}
```

Items which are larger than the $\langle \text{pivot} \rangle$ are sorted, then placed after code that sorts the smaller items, and after the (braced) $\langle \text{pivot} \rangle$.

The fourth speed up is to avoid the recursive call to `\tl_sort:nN` with an empty first argument. For this, we introduce functions similar to the `__sort_i:nnnnNn` of the last example, but aware of whether the list of $\langle \text{conditional} \rangle \{ \langle \text{item} \rangle \}$ read so far that are less than the pivot, and the list of those greater or equal, are empty or not: see `__sort_quick_split:NnNn` and functions defined below. Knowing whether the lists are empty or not is useless if we do not use distinct ending codes as appropriate. The splitting auxiliaries communicate to the $\langle \text{end-loop} \rangle$ function (that is initially placed after the “prepared” list) by placing a specific ending function, ignored when looping, but useful at the end. In fact, the $\langle \text{end-loop} \rangle$ function does nothing but place the appropriate ending function in front of all its arguments. The ending functions take care of sorting non-empty sublists, placing the pivot in between, and the continuation before.

The final change in fact slows down the code a little, but is required to avoid memory issues: schematically, when \TeX encounters

```
\use:n { \use:n { \use:n { ... } ... } ... }
```

the argument of the first `\use:n` is not completely read by the second `\use:n`, hence must remain in memory; then the argument of the second `\use:n` is not completely read when grabbing the argument of the third `\use:n`, hence must remain in memory, and so on. The memory consumption grows quadratically with the number of nested `\use:n`. In practice, this means that we must read everything until a trailing `\q_stop` once in a

while, otherwise sorting lists of more than a few thousand items would exhaust a typical \TeX 's memory.

\tl_sort:nN

$\text{_}_sort_quick_prepare:Nnnn$
 $\text{_}_sort_quick_prepare_end:NNNnw$
 $\text{_}_sort_quick_cleanup:w$

The code within the \exp_not:f sorts the list, leaving in most cases a leading \exp_not:f , which stops the expansion, letting the result be return within \exp_not:n . We filter out the case of a list with no item, which would otherwise cause problems. Then prepare the token list **#1** by inserting the conditional **#2** before each item. The **prepare** auxiliary receives the conditional as **#1**, the prepared token list so far as **#2**, the next prepared item as **#3**, and the item after that as **#4**. The loop ends when **#4** contains \prg_break_point: , then the **prepare_end** auxiliary finds the prepared token list as **#4**. The scene is then set up for $\text{_}_sort_quick_split:NnNn$, which sorts the prepared list and perform the post action placed after \q_mark , namely removing the trailing \s_stop and \q_stop and leaving \exp_stop_f: to stop f-expansion.

```

17914 \cs_new:Npn \tl_sort:nN #1#2
17915 {
17916   \exp_not:f
17917   {
17918     \tl_if_blank:nF {#1}
17919     {
17920       \_sort_quick_prepare:Nnnn #2 { } { }
17921       #1
17922       { \prg_break_point: \_sort_quick_prepare_end:NNNnw }
17923       \q_stop
17924     }
17925   }
17926 }
17927 \cs_new:Npn \_sort_quick_prepare:Nnnn #1#2#3#4
17928 {
17929   \prg_break: #4 \prg_break_point:
17930   \_sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
17931 }
17932 \cs_new:Npn \_sort_quick_prepare_end:NNNnw #1#2#3#4#5 \q_stop
17933 {
17934   \_sort_quick_split:NnNn #4 \_sort_quick_end:nnTFNn { }
17935   \q_mark { \_sort_quick_cleanup:w \exp_stop_f: }
17936   \s_stop \q_stop
17937 }
17938 \cs_new:Npn \_sort_quick_cleanup:w #1 \s_stop \q_stop {#1}

```

(End definition for \tl_sort:nN and others. This function is documented on page 43.)

$\text{_}_sort_quick_split:NnNn$

$\text{_}_sort_quick_only_i:NnnnnNn$
 $\text{_}_sort_quick_only_ii:NnnnnNn$
 $\text{_}_sort_quick_split_i:NnnnnNn$
 $\text{_}_sort_quick_split_ii:NnnnnNn$

The **only_i**, **only_ii**, **split_i** and **split_ii** auxiliaries receive a useless first argument, the new item **#2** (that they append to either one of the next two arguments), the list **#3** of items less than the pivot, bigger items **#4**, the pivot **#5**, a *function* **#6**, and an item **#7**. The *function* is the user's *conditional* except at the end of the list where it is $\text{_}_sort_quick_end:nnTFNn$. The comparison is applied to the *pivot* and the *item*, and calls the **only_i** or **split_i** auxiliaries if the *item* is smaller, and the **only_ii** or **split_ii** auxiliaries otherwise. In both cases, the next auxiliary goes to work right away, with no intermediate expansion that would slow down operations. Note that the argument **#2** left for the next call has the form *conditional* {*item*}, so that the lists **#3** and **#4** keep the right form to be fed to the next sorting function. The **split** auxiliary differs from these in that it is missing three of the arguments, which would be empty, and its first argument is always the user's *conditional* rather than an ending function.

```

17939 \cs_new:Npn \__sort_quick_split:NnNn #1#2#3#4
17940 {
17941     #3 {#2} {#4} \__sort_quick_only_ii:NnnnnNn
17942     \__sort_quick_only_i:NnnnnNn
17943     \__sort_quick_single_end:nnnwnw
17944     { #3 {#4} } { } { } {#2}
17945 }
17946 \cs_new:Npn \__sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
17947 {
17948     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
17949     \__sort_quick_only_i:NnnnnNn
17950     \__sort_quick_only_i_end:nnnwnw
17951     { #6 {#7} } { #3 #2 } { } {#5}
17952 }
17953 \cs_new:Npn \__sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
17954 {
17955     #6 {#5} {#7} \__sort_quick_only_ii:NnnnnNn
17956     \__sort_quick_split_i:NnnnnNn
17957     \__sort_quick_only_ii_end:nnnwnw
17958     { #6 {#7} } { } { #4 #2 } {#5}
17959 }
17960 \cs_new:Npn \__sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
17961 {
17962     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
17963     \__sort_quick_split_i:NnnnnNn
17964     \__sort_quick_split_end:nnnwnw
17965     { #6 {#7} } { #3 #2 } {#4} {#5}
17966 }
17967 \cs_new:Npn \__sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
17968 {
17969     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
17970     \__sort_quick_split_i:NnnnnNn
17971     \__sort_quick_split_end:nnnwnw
17972     { #6 {#7} } {#3} { #4 #2 } {#5}
17973 }

```

(End definition for __sort_quick_split:NnNn and others.)

__sort_quick_end:nnTFNn The __sort_quick_end:nnTFNn appears instead of the user's conditional, and receives as its arguments the pivot #1, a fake item #2, a **true** and a **false** branches #3 and #4, followed by an ending function #5 (one of the four auxiliaries here) and another copy #6 of the fake item. All those are discarded except the function #5. This function receives lists #1 and #2 of items less than or greater than the pivot #3, then a continuation code #5 just after \q_mark. To avoid a memory problem described earlier, all of the ending functions read #6 until \q_stop and place #6 back into the input stream. When the lists #1 and #2 are empty, the **single** auxiliary simply places the continuation #5 before the pivot {#3}. When #2 is empty, #1 is sorted and placed before the pivot {#3}, taking care to feed the continuation #5 as a continuation for the function sorting #1. When #1 is empty, #2 is sorted, and the continuation argument is used to place the continuation #5 and the pivot {#3} before the sorted result. Finally, when both lists are non-empty, items larger than the pivot are sorted, then items less than the pivot, and the continuations are done in such a way to place the pivot in between.

```

17974 \cs_new:Npn \__sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}

```

```

17975 \cs_new:Npn \__sort_quick_single_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
17976 { #5 {#3} #6 \q_stop }
17977 \cs_new:Npn \__sort_quick_only_i_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
17978 {
17979   \__sort_quick_split:NnNn #1
17980   \__sort_quick_end:nnTFNn { } \q_mark {#5}
17981   {#3}
17982   #6 \q_stop
17983 }
17984 \cs_new:Npn \__sort_quick_only_ii_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
17985 {
17986   \__sort_quick_split:NnNn #2
17987   \__sort_quick_end:nnTFNn { } \q_mark { #5 {#3} }
17988   #6 \q_stop
17989 }
17990 \cs_new:Npn \__sort_quick_split_end:nnnwnw #1#2#3#4 \q_mark #5#6 \q_stop
17991 {
17992   \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \q_mark
17993   {
17994     \__sort_quick_split:NnNn #1
17995     \__sort_quick_end:nnTFNn { } \q_mark {#5}
17996     {#3}
17997   }
17998   #6 \q_stop
17999 }

```

(End definition for __sort_quick_end:nnTFNn and others.)

35.6 Messages

__sort_error: Bailing out of the sorting code is a bit tricky. It may not be safe to use a delimited argument, so instead we redefine many l3sort commands to be trivial, with __sort_level: jumping to the break point. This error recovery won't work in a group.

```

18000 \cs_new_protected:Npn \__sort_error:
18001 {
18002   \cs_set_eq:NN \__sort_merge_blocks_aux: \prg_do_nothing:
18003   \cs_set_eq:NN \__sort_merge_blocks: \prg_do_nothing:
18004   \cs_set_protected:Npn \__sort_level: { \group_end: \prg_break: }
18005 }

```

(End definition for __sort_error:.)

__sort_disable_toksdef: While sorting, \toksdef is locally disabled to prevent users from using \newtoks or similar commands in their comparison code: the \toks registers that would be assigned are in use by l3sort. In format mode, none of this is needed since there is no \toks allocator.

```

18006 <*package>
18007 \cs_new_protected:Npn \__sort_disable_toksdef:
18008 { \cs_set_eq:NN \toksdef \__sort_disabled_toksdef:n }
18009 \cs_new_protected:Npn \__sort_disabled_toksdef:n #1
18010 {
18011   \__kernel_msg_error:nnx { kernel } { toksdef }
18012   { \token_to_str:N #1 }
18013   \__sort_error:

```

```

18014 \tex_toksdef:D #1
18015 }
18016 \__kernel_msg_new:nnnn { kernel } { toksdef }
18017 { Allocation~of~\iow_char:N\ toks~registers~impossible~while~sorting. }
18018 {
18019 The~comparison~code~used~for~sorting~a~list~has~attempted~to~
18020 define~#1~as~a~new~\iow_char:N\ toks~register~using~
18021 \iow_char:N\ newtoks~
18022 or~a~similar~command.~The~list~will~not~be~sorted.
18023 }
18024 \package

```

(End definition for __sort_disable_toksdef: and __sort_disabled_toksdef:n.)

__sort_too_long_error:NNw When there are too many items in a sequence, this is an error, and we clean up properly the mapping over items in the list: break using the type-specific breaking function #1.

```

18025 \cs_new_protected:Npn \__sort_too_long_error:NNw #1#2 \fi:
18026 {
18027 \fi:
18028 \__kernel_msg_error:nnxxx { kernel } { too-large }
18029 { \token_to_str:N #2 }
18030 { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
18031 { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
18032 #1 \__sort_error:
18033 }
18034 \__kernel_msg_new:nnnn { kernel } { too-large }
18035 { The~list~#1~is~too~long~to~be~sorted~by~TeX. }
18036 {
18037 TeX~has~#2~toks~registers~still~available:~
18038 this~only~allows~to~sort~with~up~to~#3~
18039 items.~The~list~will~not~be~sorted.
18040 }

```

(End definition for __sort_too_long_error:NNw.)

```

18041 \__kernel_msg_new:nnnn { kernel } { return-none }
18042 { The~comparison~code~did~not~return. }
18043 {
18044 When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~
18045 did~not~call~
18046 \iow_char:N\ sort_return_same: ~nor~
18047 \iow_char:N\ sort_return_swapped: .~
18048 Exactly~one~of~these~should~be~called.
18049 }
18050 \__kernel_msg_new:nnnn { kernel } { return-two }
18051 { The~comparison~code~returned~multiple~times. }
18052 {
18053 When~sorting~a~list,~the~code~to~compare~items~called~
18054 \iow_char:N\ sort_return_same: ~or~
18055 \iow_char:N\ sort_return_swapped: ~multiple~times.~
18056 Exactly~one~of~these~should~be~called.
18057 }

```


35.7 Deprecated functions

```

\sort_ordered: These functions were renamed for consistency.
\sort_reversed:
18058 \__kernel_patch_deprecation:nnNNpn { 2018-12-31 } { \sort_return_same: }
18059 \cs_new_protected:Npn \sort_ordered: { \sort_return_same: }
18060 \__kernel_patch_deprecation:nnNNpn { 2018-12-31 } { \sort_return_swapped: }
18061 \cs_new_protected:Npn \sort_reversed: { \sort_return_swapped: }

(End definition for \sort_ordered: and \sort_reversed:.)

18062 </initex | package>

```

36 l3tl-analysis implementation

```
18063 <@@=tl>
```

36.1 Internal functions

`\s__tl` The format used to store token lists internally uses the scan mark `\s__tl` as a delimiter.

(End definition for `\s__tl`.)

36.2 Internal format

The task of the `l3tl-analysis` module is to convert token lists to an internal format which allows us to extract all the relevant information about individual tokens (category code, character code), as well as reconstruct the token list quickly. This internal format is used in `l3regex` where we need to support arbitrary tokens, and it is used in conversion functions in `l3str-convert`, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any $\langle token \rangle$ (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find $\langle tokens \rangle$ which both `o`-expand and `x`-expand to the given $\langle token \rangle$. Collecting more information about the category code and character code is also useful for regular expressions, since most regexes are catcode-agnostic. The internal format thus takes the form of a succession of items of the form

$$\langle tokens \rangle \backslash s_tl \langle catcode \rangle \langle char\ code \rangle \backslash s_tl$$

The $\langle tokens \rangle$ `o`- and `x`-expand to the original token in the token list or to the cluster of tokens corresponding to one Unicode character in the given encoding (for `l3str-convert`). The $\langle catcode \rangle$ is given as a single hexadecimal digit, 0 for control sequences. The $\langle char\ code \rangle$ is given as a decimal number, -1 for control sequences.

Using delimited arguments lets us build the $\langle tokens \rangle$ progressively when doing an encoding conversion in `l3str-convert`. On the other hand, the delimiter `\s__tl` may not appear unbraced in $\langle tokens \rangle$. This is not a problem because we are careful to wrap control sequences in braces (as an argument to `\exp_not:n`) when converting from a general token list to the internal format.

The current rule for converting a $\langle token \rangle$ to a balanced set of $\langle tokens \rangle$ which both `o`-expands and `x`-expands to it is the following.

- A control sequence `\cs` becomes `\exp_not:n { \cs } \s__tl 0 -1 \s__tl`.

- A begin-group character { becomes `\exp_after:wN { \if_false: } \fi: \s__tl 1 <char code> \s__tl`.
- An end-group character } becomes `\if_false: { \fi: } \s__tl 2 <char code> \s__tl`.
- A character with any other category code becomes `\exp_not:n {<character>} \s__tl <hex catcode> <char code> \s__tl`.

18064 `<*initex | package>`

36.3 Variables and helper functions

`\s__tl` The scan mark `\s__tl` is used as a delimiter in the internal format. This is more practical than using a quark, because we would then need to control expansion much more carefully: compare `\int_value:w '#1 \s__tl` with `\int_value:w '#1 \exp_stop_f: \exp_not:N \q_mark` to extract a character code followed by the delimiter in an x-expansion.

18065 `\scan_new:N \s__tl`

(End definition for \s__tl.)

`\l__tl_analysis_token` The tokens in the token list are probed with the T_EX primitive `\futurelet`. We use `\l__tl_analysis_token` in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is `\l__tl_analysis_char_token`.

18066 `\cs_new_eq:NN \l__tl_analysis_token ?`

18067 `\cs_new_eq:NN \l__tl_analysis_char_token ?`

(End definition for \l__tl_analysis_token and \l__tl_analysis_char_token.)

`\l__tl_analysis_normal_int` The number of normal (N-type argument) tokens since the last special token.

18068 `\int_new:N \l__tl_analysis_normal_int`

(End definition for \l__tl_analysis_normal_int.)

`\l__tl_analysis_index_int` During the first pass, this is the index in the array being built. During the second pass, it is equal to the maximum index in the array from the first pass.

18069 `\int_new:N \l__tl_analysis_index_int`

(End definition for \l__tl_analysis_index_int.)

`\l__tl_analysis_nesting_int` Nesting depth of explicit begin-group and end-group characters during the first pass. This lets us detect the end of the token list without a reserved end-marker.

18070 `\int_new:N \l__tl_analysis_nesting_int`

(End definition for \l__tl_analysis_nesting_int.)

`\l__tl_analysis_type_int` When encountering special characters, we record their “type” in this integer.

18071 `\int_new:N \l__tl_analysis_type_int`

(End definition for \l__tl_analysis_type_int.)

`\g__tl_analysis_result_tl` The result of the conversion is stored in this token list, with a succession of items of the form

$\langle tokens \rangle \backslash s_tl \langle catcode \rangle \langle char code \rangle \backslash s_tl$

18072 $\backslash tl_new:N \backslash g_tl_analysis_result_tl$

(End definition for $\backslash g_tl_analysis_result_tl$.)

$\backslash_tl_analysis_extract_charcode:$ Extracting the character code from the meaning of $\backslash l_tl_analysis_token$. This has no error checking, and should only be assumed to work for begin-group and end-group character tokens. It produces a number in the form ‘ $\langle char \rangle$ ’.

18073 $\backslash cs_new:Npn \backslash_tl_analysis_extract_charcode:$

18074 {

18075 $\backslash exp_after:wN \backslash_tl_analysis_extract_charcode_aux:w$

18076 $\backslash token_to_meaning:N \backslash l_tl_analysis_token$

18077 }

18078 $\backslash cs_new:Npn \backslash_tl_analysis_extract_charcode_aux:w \#1 \sim \#2 \sim \{ ' \}$

(End definition for $\backslash_tl_analysis_extract_charcode:$ and $\backslash_tl_analysis_extract_charcode_aux:w$.)

$\backslash_tl_analysis_cs_space_count:NN$ Counts the number of spaces in the string representation of its second argument, as well as the number of characters following the last space in that representation, and feeds the two numbers as semicolon-delimited arguments to the first argument. When this function is used, the escape character is printable and non-space.

18079 $\backslash cs_new:Npn \backslash_tl_analysis_cs_space_count:NN \#1 \#2$

18080 {

18081 $\backslash exp_after:wN \#1$

18082 $\backslash int_value:w \backslash int_eval:w 0$

18083 $\backslash exp_after:wN \backslash_tl_analysis_cs_space_count:w$

18084 $\backslash token_to_str:N \#2$

18085 $\backslash fi: \backslash_tl_analysis_cs_space_count_end:w ; \sim !$

18086 }

18087 $\backslash cs_new:Npn \backslash_tl_analysis_cs_space_count:w \#1 \sim$

18088 {

18089 $\backslash if_false: \#1 \#1 \backslash fi:$

18090 $+ 1$

18091 $\backslash_tl_analysis_cs_space_count:w$

18092 }

18093 $\backslash cs_new:Npn \backslash_tl_analysis_cs_space_count_end:w ; \#1 \backslash fi: \#2 !$

18094 { $\backslash exp_after:wN ; \backslash int_value:w \backslash str_count_ignore_spaces:n \{ \#1 \} ;$ }

(End definition for $\backslash_tl_analysis_cs_space_count:NN$, $\backslash_tl_analysis_cs_space_count:w$, and $\backslash_tl_analysis_cs_space_count_end:w$.)

36.4 Plan of attack

Our goal is to produce a token list of the form roughly

$\langle token\ 1 \rangle \backslash s@ \langle catcode\ 1 \rangle \langle char\ code\ 1 \rangle \backslash s@$
 $\langle token\ 2 \rangle \backslash s_tl \langle catcode\ 2 \rangle \langle char\ code\ 2 \rangle \backslash s_tl$
 $\dots \langle token\ N \rangle \backslash s_tl \langle catcode\ N \rangle \langle char\ code\ N \rangle \backslash s_tl$

Most but not all tokens can be grabbed as an undelimited (N-type) argument by T_EX. The plan is to have a two pass system. In the first pass, locate special tokens, and store them in various $\backslash toks$ registers. In the second pass, which is done within an x-expanding assignment, normal tokens are taken in as N-type arguments, and special tokens are

retrieved from the `\toks` registers, and removed from the input stream by some means. The whole process takes linear time, because we avoid building the result one item at a time.

We make the escape character printable (backslash, but this later oscillates between slash and backslash): this allows us to distinguish characters from control sequences.

A token has two characteristics: its `\meaning`, and what it looks like for `TEX` when it is in scanning mode (*e.g.*, when capturing parameters for a macro). For our purposes, we distinguish the following meanings:

- begin-group token (category code 1), either space (character code 32), or non-space;
- end-group token (category code 2), either space (character code 32), or non-space;
- space token (category code 10, character code 32);
- anything else (then the token is always an N-type argument).

The token itself can “look like” one of the following

- a non-active character, in which case its meaning is automatically that associated to its character code and category code, we call it “true” character;
- an active character;
- a control sequence.

The only tokens which are not valid N-type arguments are true begin-group characters, true end-group characters, and true spaces. We detect those characters by scanning ahead with `\futurelet`, then distinguishing true characters from control sequences set equal to them using the `\string` representation.

The second pass is a simple exercise in expandable loops.

`__tl_analysis:n` Everything is done within a group, and all definitions are local. We use `\group_align_safe_begin/end:` to avoid problems in case `__tl_analysis:n` is used within an alignment and its argument contains alignment tab tokens.

```

18095 \cs_new_protected:Npn \__tl_analysis:n #1
18096 {
18097   \group_begin:
18098     \group_align_safe_begin:
18099       \__tl_analysis_a:n {#1}
18100       \__tl_analysis_b:n {#1}
18101     \group_align_safe_end:
18102   \group_end:
18103 }
```

(End definition for `__tl_analysis:n`.)

36.5 Disabling active characters

`__tl_analysis_disable:n` Active characters can cause problems later on in the processing, so we provide a way to disable them, by setting them to `undefined`. Since Unicode contains too many characters to loop over all of them, we instead do this whenever we encounter a character. For `pTeX` and `upTeX` we skip characters beyond `[0, 255]` because `\lccode` only allows those values.

```
18104 \group_begin:
18105   \char_set_catcode_active:N \^^@
18106   \cs_new_protected:Npn \__tl_analysis_disable:n #1
18107     {
18108       \tex_lccode:D 0 = #1 \exp_stop_f:
18109       \tex_lowercase:D { \tex_let:D \^^@ } \tex_undefined:D
18110     }
18111   \bool_lazy_or:nnT
18112     { \sys_if_engine_ptex_p: }
18113     { \sys_if_engine_uptex_p: }
18114     {
18115       \cs_gset_protected:Npn \__tl_analysis_disable:n #1
18116         {
18117           \if_int_compare:w 256 > #1 \exp_stop_f:
18118           \tex_lccode:D 0 = #1 \exp_stop_f:
18119           \tex_lowercase:D { \tex_let:D \^^@ } \tex_undefined:D
18120         }
18121     }
18122   }
18123 \group_end:
```

(End definition for `__tl_analysis_disable:n`.)

36.6 First pass

The goal of this pass is to detect special (non-N-type) tokens, and count how many N-type tokens lie between special tokens. Also, we wish to store some representation of each special token in a `\toks` register.

We have 11 types of tokens:

1. a true non-space begin-group character;
2. a true space begin-group character;
3. a true non-space end-group character;
4. a true space end-group character;
5. a true space blank space character;
6. an active character;
7. any other true character;
8. a control sequence equal to a begin-group token (category code 1);
9. a control sequence equal to an end-group token (category code 2);
10. a control sequence equal to a space token (character code 32, category code 10);

11. any other control sequence.

Our first tool is `\futurelet`. This cannot distinguish case 8 from 1 or 2, nor case 9 from 3 or 4, nor case 10 from case 5. Those cases are later distinguished by applying the `\string` primitive to the following token, after possibly changing the escape character to ensure that a control sequence’s string representation cannot be mistaken for the true character.

In cases 6, 7, and 11, the following token is a valid N-type argument, so we grab it and distinguish the case of a character from a control sequence: in the latter case, `\str_tail:n {\token}` is non-empty, because the escape character is printable.

`__tl_analysis_a:n` We read tokens one by one using `\futurelet`. While performing the loop, we keep track of the number of true begin-group characters minus the number of true end-group characters in `\l__tl_analysis_nesting_int`. This reaches -1 when we read the closing brace.

```
18124 \cs_new_protected:Npn \__tl_analysis_a:n #1
18125 {
18126   \__tl_analysis_disable:n { 32 }
18127   \int_set:Nn \tex_escapechar:D { 92 }
18128   \int_zero:N \l__tl_analysis_normal_int
18129   \int_zero:N \l__tl_analysis_index_int
18130   \int_zero:N \l__tl_analysis_nesting_int
18131   \if_false: { \fi: \__tl_analysis_a_loop:w #1 }
18132   \int_decr:N \l__tl_analysis_index_int
18133 }
```

(End definition for `__tl_analysis_a:n`.)

`__tl_analysis_a_loop:w` Read one character and check its type.

```
18134 \cs_new_protected:Npn \__tl_analysis_a_loop:w
18135 { \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_type:w }
```

(End definition for `__tl_analysis_a_loop:w`.)

`__tl_analysis_a_type:w` At this point, `\l__tl_analysis_token` holds the meaning of the following token. We store in `\l__tl_analysis_type_int` information about the meaning of the token ahead:

- 0 space token;
- 1 begin-group token;
- -1 end-group token;
- 2 other.

The values 0, 1, -1 correspond to how much a true such character changes the nesting level (2 is used only here, and is irrelevant later). Then call the auxiliary for each case. Note that nesting conditionals here is safe because we only skip over `\l__tl_analysis_token` if it matches with one of the character tokens (hence is not a primitive conditional).

```
18136 \cs_new_protected:Npn \__tl_analysis_a_type:w
18137 {
18138   \l__tl_analysis_type_int =
18139   \if_meaning:w \l__tl_analysis_token \c_space_token
18140     0
```

```

18141     \else:
18142         \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
18143             1
18144         \else:
18145             \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
18146                 - 1
18147             \else:
18148                 2
18149             \fi:
18150         \fi:
18151     \fi:
18152     \exp_stop_f:
18153     \if_case:w \l__tl_analysis_type_int
18154         \exp_after:wN \__tl_analysis_a_space:w
18155     \or: \exp_after:wN \__tl_analysis_a_bgroup:w
18156     \or: \exp_after:wN \__tl_analysis_a_safe:N
18157     \else: \exp_after:wN \__tl_analysis_a_egroup:w
18158     \fi:
18159 }

```

(End definition for __tl_analysis_a_type:w.)

__tl_analysis_a_space:w
 __tl_analysis_a_space_test:w

In this branch, the following token's meaning is a blank space. Apply \string to that token: a true blank space gives a space, a control sequence gives a result starting with the escape character, an active character gives something else than a space since we disabled the space. We grab as \l__tl_analysis_char_token the first character of the string representation then test it in __tl_analysis_a_space_test:w. Also, since __tl_analysis_a_store: expects the special token to be stored in the relevant \toks register, we do that. The extra \exp_not:n is unnecessary of course, but it makes the treatment of all tokens more homogeneous. If we discover that the next token was actually a control sequence or an active character instead of a true space, then we step the counter of normal tokens. We now have in front of us the whole string representation of the control sequence, including potential spaces; those will appear to be true spaces later in this pass. Hence, all other branches of the code in this first pass need to consider the string representation, so that the second pass does not need to test the meaning of tokens, only strings.

```

18160 \cs_new_protected:Npn \__tl_analysis_a_space:w
18161 {
18162     \tex_afterassignment:D \__tl_analysis_a_space_test:w
18163     \exp_after:wN \cs_set_eq:NN
18164     \exp_after:wN \l__tl_analysis_char_token
18165     \token_to_str:N
18166 }
18167 \cs_new_protected:Npn \__tl_analysis_a_space_test:w
18168 {
18169     \if_meaning:w \l__tl_analysis_char_token \c_space_token
18170         \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
18171         \__tl_analysis_a_store:
18172     \else:
18173         \int_incr:N \l__tl_analysis_normal_int
18174     \fi:
18175     \__tl_analysis_a_loop:w
18176 }

```

(End definition for `_tl_analysis_a_space:w` and `_tl_analysis_a_space_test:w`.)

`_tl_analysis_a_bgroup:w` The token is most likely a true character token with catcode 1 or 2, but it might be a control sequence, or an active character. Optimizing for the first case, we store in a toks register some code that expands to that token. Since we will turn what follows into a string, we make sure the escape character is different from the current character code (by switching between solidus and backslash). To detect the special case of an active character let to the catcode 1 or 2 character with the same character code, we disable the active character with that character code and re-test: if the following token has become undefined we can in fact safely grab it. We are finally ready to turn what follows to a string and test it. This is one place where we need `\l_tl_analysis_char_token` to be a separate control sequence from `\l_tl_analysis_token`, to compare them.

```

18177 \group_begin:
18178   \char_set_catcode_group_begin:N \^^@ % {
18179   \cs_new_protected:Npn \_tl\_analysis\_a\_bgroup:w
18180     { \_tl\_analysis\_a\_group:nw { \exp_after:wN \^^@ \if_false: } \fi: } }
18181   \char_set_catcode_group_end:N \^^@
18182   \cs_new_protected:Npn \_tl\_analysis\_a\_egroup:w
18183     { \_tl\_analysis\_a\_group:nw { \if_false: { \fi: \^^@ } } % }
18184 \group_end:
18185 \cs_new_protected:Npn \_tl\_analysis\_a\_group:nw #1
18186 {
18187   \tex_lccode:D 0 = \_tl\_analysis\_extract\_charcode: \scan_stop:
18188   \tex_lowercase:D { \tex_toks:D \l\_tl\_analysis\_index\_int {#1} }
18189   \if_int_compare:w \tex_lccode:D 0 = \tex_escapechar:D
18190     \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }
18191   \fi:
18192   \_tl\_analysis\_disable:n { \tex_lccode:D 0 }
18193   \tex_futurelet:D \l\_tl\_analysis\_token \_tl\_analysis\_a\_group\_aux:w
18194 }
18195 \cs_new_protected:Npn \_tl\_analysis\_a\_group\_aux:w
18196 {
18197   \if_meaning:w \l\_tl\_analysis\_token \tex_undefined:D
18198     \exp_after:wN \_tl\_analysis\_a\_safe:N
18199   \else:
18200     \exp_after:wN \_tl\_analysis\_a\_group\_auxii:w
18201   \fi:
18202 }
18203 \cs_new_protected:Npn \_tl\_analysis\_a\_group\_auxii:w
18204 {
18205   \tex_afterassignment:D \_tl\_analysis\_a\_group\_test:w
18206   \exp_after:wN \cs_set_eq:NN
18207   \exp_after:wN \l\_tl\_analysis\_char\_token
18208   \token_to_str:N
18209 }
18210 \cs_new_protected:Npn \_tl\_analysis\_a\_group\_test:w
18211 {
18212   \if_charcode:w \l\_tl\_analysis\_token \l\_tl\_analysis\_char\_token
18213     \_tl\_analysis\_a\_store:
18214   \else:
18215     \int_incr:N \l\_tl\_analysis\_normal\_int
18216   \fi:
18217   \_tl\_analysis\_a\_loop:w

```



```
18218 }
```

(End definition for _tl_analysis_a_bgroup:w and others.)

`_tl_analysis_a_store:` This function is called each time we meet a special token; at this point, the `\toks` register `\l_tl_analysis_index_int` holds a token list which expands to the given special token. Also, the value of `\l_tl_analysis_type_int` indicates which case we are in:

- -1 end-group character;
- 0 space character;
- 1 begin-group character.

We need to distinguish further the case of a space character (code 32) from other character codes, because those behave differently in the second pass. Namely, after testing the `\lccode` of 0 (which holds the present character code) we change the cases above to

- -2 space end-group character;
- -1 non-space end-group character;
- 0 space blank space character;
- 1 non-space begin-group character;
- 2 space begin-group character.

This has the property that non-space characters correspond to odd values of `\l_tl_analysis_type_int`. The number of normal tokens until here and the type of special token are packed into a `\skip` register. Finally, we check whether we reached the last closing brace, in which case we stop by disabling the looping function (locally).

```
18219 \cs_new_protected:Npn \_tl\_analysis\_a\_store:
18220 {
18221   \tex_advance:D \l\_tl\_analysis\_nesting\_int \l\_tl\_analysis\_type\_int
18222   \if_int_compare:w \tex_lccode:D 0 = '\ \exp_stop_f:
18223     \tex_advance:D \l\_tl\_analysis\_type\_int \l\_tl\_analysis\_type\_int
18224   \fi:
18225   \tex_skip:D \l\_tl\_analysis\_index\_int
18226     = \l\_tl\_analysis\_normal\_int sp
18227     plus \l\_tl\_analysis\_type\_int sp \scan_stop:
18228   \int_incr:N \l\_tl\_analysis\_index\_int
18229   \int_zero:N \l\_tl\_analysis\_normal\_int
18230   \if_int_compare:w \l\_tl\_analysis\_nesting\_int = -1 \exp_stop_f:
18231     \cs_set_eq:NN \_tl\_analysis\_a\_loop:w \scan_stop:
18232   \fi:
18233 }
```

(End definition for _tl_analysis_a_store:.)

`_tl_analysis_a_safe:N` This should be the simplest case: since the upcoming token is safe, we can simply grab it in a second pass. If the token is a single character (including space), the `\if_charcode:w` test yields true; we disable a potentially active character (that could otherwise masquerade as the true character in the next pass) and we count one “normal” token. On the other hand, if the token is a control sequence, we should replace it by its string representation for compatibility with other code branches. Instead of slowly looping through

the characters with the main code, we use the knowledge of how the second pass works: if the control sequence name contains no space, count that token as a number of normal tokens equal to its string length. If the control sequence contains spaces, they should be registered as special characters by increasing `\l__tl_analysis_index_int` (no need to carefully count character between each space), and all characters after the last space should be counted in the following sequence of “normal” tokens.

```

18234 \cs_new_protected:Npn \__tl_analysis_a_safe:N #1
18235 {
18236   \if_charcode:w
18237     \scan_stop:
18238     \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
18239     \scan_stop:
18240     \exp_after:wN \use_i:nn
18241   \else:
18242     \exp_after:wN \use_ii:nn
18243   \fi:
18244   {
18245     \__tl_analysis_disable:n { '#1 }
18246     \int_incr:N \l__tl_analysis_normal_int
18247   }
18248   { \__tl_analysis_cs_space_count:NN \__tl_analysis_a_cs:ww #1 }
18249   \__tl_analysis_a_loop:w
18250 }
18251 \cs_new_protected:Npn \__tl_analysis_a_cs:ww #1; #2;
18252 {
18253   \if_int_compare:w #1 > 0 \exp_stop_f:
18254     \tex_skip:D \l__tl_analysis_index_int
18255     = \int_eval:n { \l__tl_analysis_normal_int + 1 } sp \exp_stop_f:
18256     \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
18257   \else:
18258     \tex_advance:D
18259   \fi:
18260   \l__tl_analysis_normal_int #2 \exp_stop_f:
18261 }

```

(End definition for `__tl_analysis_a_safe:N` and `__tl_analysis_a_cs:ww`.)

36.7 Second pass

The second pass is an exercise in expandable loops. All the necessary information is stored in `\skip` and `\toks` registers.

```

\__tl_analysis_b:n
\__tl_analysis_b_loop:w

```

Start the loop with the index 0. No need for an end-marker: the loop stops by itself when the last index is read. We repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

```

18262 \cs_new_protected:Npn \__tl_analysis_b:n #1
18263 {
18264   \tl_gset:Nx \g__tl_analysis_result_tl
18265   {
18266     \__tl_analysis_b_loop:w 0; #1
18267     \prg_break_point:
18268   }
18269 }

```

```

18270 \cs_new:Npn \__tl_analysis_b_loop:w #1;
18271 {
18272   \exp_after:wN \__tl_analysis_b_normals:ww
18273   \int_value:w \tex_skip:D #1 ; #1 ;
18274 }

```

(End definition for __tl_analysis_b:n and __tl_analysis_b_loop:w.)

__tl_analysis_b_normals:ww
__tl_analysis_b_normal:wwN

The first argument is the number of normal tokens which remain to be read, and the second argument is the index in the array produced in the first step. A character's string representation is always one character long, while a control sequence is always longer (we have set the escape character to a printable value). In both cases, we leave \exp_not:n {⟨token⟩} \s__tl in the input stream (after x-expansion). Here, \exp_not:n is used rather than \exp_not:N because #3 could be a macro parameter character or could be \s__tl (which must be hidden behind braces in the result).

```

18275 \cs_new:Npn \__tl_analysis_b_normals:ww #1;
18276 {
18277   \if_int_compare:w #1 = 0 \exp_stop_f:
18278   \__tl_analysis_b_special:w
18279   \fi:
18280   \__tl_analysis_b_normal:wwN #1;
18281 }
18282 \cs_new:Npn \__tl_analysis_b_normal:wwN #1; #2; #3
18283 {
18284   \exp_not:n { \exp_not:n { #3 } } \s__tl
18285   \if_charcode:w
18286     \scan_stop:
18287     \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
18288     \scan_stop:
18289     \exp_after:wN \__tl_analysis_b_char:Nww
18290   \else:
18291     \exp_after:wN \__tl_analysis_b_cs:Nww
18292   \fi:
18293   #3 #1; #2;
18294 }

```

(End definition for __tl_analysis_b_normals:ww and __tl_analysis_b_normal:wwN.)

__tl_analysis_b_char:Nww

If the normal token we grab is a character, leave ⟨catcode⟩ ⟨charcode⟩ followed by \s__tl in the input stream, and call __tl_analysis_b_normals:ww with its first argument decremented.

```

18295 \cs_new:Npx \__tl_analysis_b_char:Nww #1
18296 {
18297   \exp_not:N \if_meaning:w #1 \exp_not:N \tex_undefined:D
18298   \token_to_str:N D \exp_not:N \else:
18299   \exp_not:N \if_catcode:w #1 \c_catcode_other_token
18300   \token_to_str:N C \exp_not:N \else:
18301   \exp_not:N \if_catcode:w #1 \c_catcode_letter_token
18302   \token_to_str:N B \exp_not:N \else:
18303   \exp_not:N \if_catcode:w #1 \c_math_toggle_token      3
18304   \exp_not:N \else:
18305   \exp_not:N \if_catcode:w #1 \c_alignment_token      4
18306   \exp_not:N \else:
18307   \exp_not:N \if_catcode:w #1 \c_math_superscript_token 7

```

```

18308     \exp_not:N \else:
18309 \exp_not:N \if_catcode:w #1 \c_math_subscript_token 8
18310     \exp_not:N \else:
18311 \exp_not:N \if_catcode:w #1 \c_space_token
18312     \token_to_str:N A \exp_not:N \else:
18313     6
18314 \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: }
18315 \exp_not:N \int_value:w '#1 \s__tl
18316 \exp_not:N \exp_after:wN \exp_not:N \__tl_analysis_b_normals:ww
18317     \exp_not:N \int_value:w \exp_not:N \int_eval:w - 1 +
18318 }

```

(End definition for __tl_analysis_b_char:Nww.)

__tl_analysis_b_cs:Nww If the token we grab is a control sequence, leave 0 -1 (as category code and character code) in the input stream, followed by \s__tl, and call __tl_analysis_b_normals:ww with updated arguments.

```

18319 \cs_new:Npn \__tl_analysis_b_cs:Nww #1
18320 {
18321     0 -1 \s__tl
18322     \__tl_analysis_cs_space_count:NN \__tl_analysis_b_cs_test:ww #1
18323 }
18324 \cs_new:Npn \__tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
18325 {
18326     \exp_after:wN \__tl_analysis_b_normals:ww
18327     \int_value:w \int_eval:w
18328     \if_int_compare:w #1 = 0 \exp_stop_f:
18329     #3
18330     \else:
18331         \tex_skip:D \int_eval:n { #4 + #1 } \exp_stop_f:
18332     \fi:
18333     - #2
18334     \exp_after:wN ;
18335     \int_value:w \int_eval:n { #4 + #1 } ;
18336 }

```

(End definition for __tl_analysis_b_cs:Nww and __tl_analysis_b_cs_test:ww.)

__tl_analysis_b_special:w Here, #1 is the current index in the array built in the first pass. Check now whether we reached the end (we shouldn't keep the trailing end-group character that marked the end of the token list in the first pass). Unpack the \toks register: when x-expanding again, we will get the special token. Then leave the category code in the input stream, followed by the character code, and call __tl_analysis_b_loop:w with the next index.

```

18337 \group_begin:
18338     \char_set_catcode_other:N A
18339     \cs_new:Npn \__tl_analysis_b_special:w
18340         \fi: \__tl_analysis_b_normal:wwN 0 ; #1 ;
18341     {
18342         \fi:
18343         \if_int_compare:w #1 = \l__tl_analysis_index_int
18344             \exp_after:wN \prg_break:
18345         \fi:
18346         \tex_the:D \tex_toks:D #1 \s__tl
18347         \if_case:w \etex_gluestretch:D \tex_skip:D #1 \exp_stop_f:

```

```

18348         \token_to_str:N A
18349     \or: 1
18350     \or: 1
18351     \else: 2
18352     \fi:
18353     \if_int_odd:w \etex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
18354         \exp_after:wN \_tl_analysis_b_special_char:wN \int_value:w
18355     \else:
18356         \exp_after:wN \_tl_analysis_b_special_space:w \int_value:w
18357     \fi:
18358     \int_eval:n { 1 + #1 } \exp_after:wN ;
18359     \token_to_str:N
18360 }
18361 \group_end:
18362 \cs_new:Npn \_tl_analysis_b_special_char:wN #1 ; #2
18363 {
18364     \int_value:w '#2 \s_tl
18365     \_tl_analysis_b_loop:w #1 ;
18366 }
18367 \cs_new:Npn \_tl_analysis_b_special_space:w #1 ; ~
18368 {
18369     32 \s_tl
18370     \_tl_analysis_b_loop:w #1 ;
18371 }

```

(End definition for `_tl_analysis_b_special:w`, `_tl_analysis_b_special_char:wN`, and `_tl_analysis_b_special_space:w`.)

36.8 Mapping through the analysis

```

\tl_analysis_map_inline:nn
\tl_analysis_map_inline:Nn
  \_tl_analysis_map_inline_aux:Nn
  \_tl_analysis_map_inline_aux:nnn

```

First obtain the analysis of the token list into `\g__tl_analysis_result_tl`. To allow nested mappings, increase the nesting depth `\g__kernel_prg_map_int` (shared between all modules), then define the looping macro, which has a name specific to that nesting depth. That looping grabs the `<tokens>`, `<catcode>` and `<char code>`; it checks for the end of the loop with `\use_none:n ##2`, normally empty, but which becomes `\tl_map_break:` at the end; it then performs the user's code `#2`, and loops by calling itself. When the loop ends, remember to decrease the nesting depth.

```

18372 \cs_new_protected:Npn \tl_analysis_map_inline:nn #1
18373 {
18374     \_tl_analysis:n {#1}
18375     \int_gincr:N \g__kernel_prg_map_int
18376     \exp_args:Nc \_tl_analysis_map_inline_aux:Nn
18377         { \_tl_analysis_map_inline_ \int_use:N \g__kernel_prg_map_int :wNw }
18378 }
18379 \cs_new_protected:Npn \tl_analysis_map_inline:Nn #1
18380 { \exp_args:No \tl_analysis_map_inline:nn #1 }
18381 \cs_new_protected:Npn \_tl_analysis_map_inline_aux:Nn #1#2
18382 {
18383     \cs_gset_protected:Npn #1 ##1 \s_tl ##2 ##3 \s_tl
18384     {
18385         \use_none:n ##2
18386         \_tl_analysis_map_inline_aux:nnn {##1} {##3} {##2}
18387     }
18388     \cs_gset_protected:Npn \_tl_analysis_map_inline_aux:nnn ##1##2##3

```

```

18389     {
18390         #2
18391         #1
18392     }
18393     \exp_after:wN #1
18394     \g__tl_analysis_result_tl
18395     \s__tl { ? \tl_map_break: } \s__tl
18396     \prg_break_point:Nn \tl_map_break:
18397     { \int_gdecr:N \g__kernel_prg_map_int }
18398 }

```

(End definition for `\tl_analysis_map_inline:nn` and others. These functions are documented on page 201.)

36.9 Showing the results

`\tl_analysis_show:N` Add to `__tl_analysis:n` a third pass to display tokens to the terminal. If the token list variable is not defined, throw the same error as `\tl_show:N` by simply calling that function.

```

18399 \cs_new_protected:Npn \tl_analysis_show:N #1
18400 {
18401     \tl_if_exist:NTF #1
18402     {
18403         \exp_args:No \__tl_analysis:n {#1}
18404         \msg_show:nnxxxx { LaTeX / kernel } { show-tl-analysis }
18405         { \token_to_str:N #1 } { \__tl_analysis_show: } { } { }
18406     }
18407     { \tl_show:N #1 }
18408 }
18409 \cs_new_protected:Npn \tl_analysis_show:n #1
18410 {
18411     \__tl_analysis:n {#1}
18412     \msg_show:nnxxxx { LaTeX / kernel } { show-tl-analysis }
18413     { } { \__tl_analysis_show: } { } { }
18414 }

```

(End definition for `\tl_analysis_show:N` and `\tl_analysis_show:n`. These functions are documented on page 201.)

`__tl_analysis_show:` Here, `#1` o- and x-expands to the token; `#2` is the category code (one uppercase hexadecimal digit), 0 for control sequences; `#3` is the character code, which we ignore. In the cases of control sequences and active characters, the meaning may overflow one line, and we want to truncate it. Those cases are thus separated out.

```

18415 \cs_new:Npn \__tl_analysis_show:
18416 {
18417     \exp_after:wN \__tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
18418     \s__tl { ? \prg_break: } \s__tl
18419     \prg_break_point:
18420 }
18421 \cs_new:Npn \__tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
18422 {
18423     \use_none:n #2
18424     \iow_newline: > \use:nn { ~ } { ~ }
18425     \if_int_compare:w "#2 = 0 \exp_stop_f:

```

```

18426     \exp_after:wN \_tl_analysis_show_cs:n
18427 \else:
18428     \if_int_compare:w "#2 = 13 \exp_stop_f:
18429         \exp_after:wN \exp_after:wN
18430         \exp_after:wN \_tl_analysis_show_active:n
18431     \else:
18432         \exp_after:wN \exp_after:wN
18433         \exp_after:wN \_tl_analysis_show_normal:n
18434     \fi:
18435 \fi:
18436 {#1}
18437 \_tl_analysis_show_loop:wNw
18438 }

```

(End definition for _tl_analysis_show: and _tl_analysis_show_loop:wNw.)

_tl_analysis_show_normal:n Non-active characters are a simple matter of printing the character, and its meaning. Our test suite checks that begin-group and end-group characters do not mess up T_EX's alignment status.

```

18439 \cs_new:Npn \_tl_analysis_show_normal:n #1
18440 {
18441     \exp_after:wN \token_to_str:N #1 ~
18442     ( \exp_after:wN \token_to_meaning:N #1 )
18443 }

```

(End definition for _tl_analysis_show_normal:n.)

_tl_analysis_show_value:N This expands to the value of #1 if it has any.

```

18444 \cs_new:Npn \_tl_analysis_show_value:N #1
18445 {
18446     \token_if_expandable:NF #1
18447     {
18448         \token_if_chardef:NTF #1 \prg_break: { }
18449         \token_if_mathchardef:NTF #1 \prg_break: { }
18450         \token_if_dim_register:NTF #1 \prg_break: { }
18451         \token_if_int_register:NTF #1 \prg_break: { }
18452         \token_if_skip_register:NTF #1 \prg_break: { }
18453         \token_if_toks_register:NTF #1 \prg_break: { }
18454         \use_none:nnn
18455         \prg_break_point:
18456         \use:n { \exp_after:wN = \tex_the:D #1 }
18457     }
18458 }

```

(End definition for _tl_analysis_show_value:N.)

_tl_analysis_show_cs:n Control sequences and active characters are printed in the same way, making sure not to go beyond the \l_iow_line_count_int. In case of an overflow, we replace the last characters by \c_tl_analysis_show_etc_str.

```

\_tl_analysis_show_long:nnn
\_tl_analysis_show_long_aux:nnnn
18459 \cs_new:Npn \_tl_analysis_show_cs:n #1
18460 { \exp_args:No \_tl_analysis_show_long:nn {#1} { control~sequence= } }
18461 \cs_new:Npn \_tl_analysis_show_active:n #1
18462 { \exp_args:No \_tl_analysis_show_long:nn {#1} { active~character= } }
18463 \cs_new:Npn \_tl_analysis_show_long:nn #1

```

```

18464 {
18465   \__tl_analysis_show_long_aux:oofn
18466   { \token_to_str:N #1 }
18467   { \token_to_meaning:N #1 }
18468   { \__tl_analysis_show_value:N #1 }
18469 }
18470 \cs_new:Npn \__tl_analysis_show_long_aux:nnnn #1#2#3#4
18471 {
18472   \int_compare:nNnTF
18473     { \str_count:n { #1 ~ ( #4 #2 #3 ) } }
18474     > { \l_iow_line_count_int - 3 }
18475     {
18476       \str_range:nnn { #1 ~ ( #4 #2 #3 ) } { 1 }
18477       {
18478         \l_iow_line_count_int - 3
18479         - \str_count:N \c__tl_analysis_show_etc_str
18480       }
18481       \c__tl_analysis_show_etc_str
18482     }
18483     { #1 ~ ( #4 #2 #3 ) }
18484 }
18485 \cs_generate_variant:Nn \__tl_analysis_show_long_aux:nnnn { oof }

```

(End definition for __tl_analysis_show_cs:n and others.)

36.10 Messages

\c__tl_analysis_show_etc_str When a control sequence (or active character) and its meaning are too long to fit in one line of the terminal, the end is replaced by this token list.

```

18486 \tl_const:Nx \c__tl_analysis_show_etc_str % (
18487   { \token_to_str:N \ETC.) }

```

(End definition for \c__tl_analysis_show_etc_str.)

```

18488 \__kernel_msg_new:nnn { kernel } { show-tl-analysis }
18489 {
18490   The-token-list~ \tl_if_empty:nF {#1} { #1 ~ }
18491   \tl_if_empty:nTF {#2}
18492     { is-empty }
18493     { contains-the-tokens: #2 }
18494 }

```

36.11 Deprecated functions

\tl_show_analysis:N Simple renames.

```

\tl_show_analysis:n
18495 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 }
18496 { \tl_analysis_show:N }
18497 \cs_new_protected:Npn \tl_show_analysis:N #1
18498   { \tl_analysis_show:N #1 }
18499 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 }
18500 { \tl_analysis_show:n }
18501 \cs_new_protected:Npn \tl_show_analysis:n #1
18502   { \tl_analysis_show:n {#1} }

```


(End definition for `\tl_show_analysis:N` and `\tl_show_analysis:n`.)

18503 `</initex | package>`

37 l3regex implementation

18504 `<*initex | package>`

18505 `<@@=regex>`

37.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (back-references come to mind first), but it is costly, and raises the problem of catastrophic backtracking. Since \TeX is not first and foremost a programming language, complicated code tends to run slowly, and we must use faster, albeit slightly more restrictive, techniques, coming from automata theory.

Given a regular expression of n characters, we do the following:

- (Compiling.) Analyse the regex, finding invalid input, and convert it to an internal representation.
- (Building.) Convert the compiled regex to a non-deterministic finite automaton (NFA) with $O(n)$ states which accepts precisely token lists matching that regex.
- (Matching.) Loop through the query token list one token (one “position”) at a time, exploring in parallel every possible path (“active thread”) through the NFA, considering active threads in an order determined by the quantifiers’ greediness.

We use the following vocabulary in the code comments (and in variable names).

- *Group*: index of the capturing group, -1 for non-capturing groups.
- *Position*: each token in the query is labelled by an integer $\langle position \rangle$, with $\text{min_pos} - 1 \leq \langle position \rangle \leq \text{max_pos}$. The lowest and highest positions correspond to imaginary begin and end markers (with inaccessible category code and character code).
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer $\langle state \rangle$ with $\text{min_state} \leq \langle state \rangle < \text{max_state}$.
- *Active thread*: state of the NFA that is reached when reading the query token list for the matching. Those threads are ordered according to the greediness of quantifiers.
- *Step*: used when matching, starts at 0, incremented every time a character is read, and is not reset when searching for repeated matches. The integer `\l__regex_step_int` is a unique id for all the steps of the matching algorithm.

We use `l3intarray` to manipulate arrays of integers (stored into some dimension registers in scaled points). We also abuse \TeX ’s `\toks` registers, by accessing them directly by number rather than tying them to control sequence using the `\newtoks` allocation functions. Specifically, these arrays and `\toks` are used as follows. When building, `\toks\langle state \rangle` holds the tests and actions to perform in the $\langle state \rangle$ of the NFA. When matching,

- `\g__regex_state_active_intarray` holds the last $\langle step \rangle$ in which each $\langle state \rangle$ was active.
- `\g__regex_thread_state_intarray` maps each $\langle thread \rangle$ (with $\text{min_active} \leq \langle thread \rangle < \text{max_active}$) to the $\langle state \rangle$ in which the $\langle thread \rangle$ currently is. The $\langle threads \rangle$ or ordered starting from the best to the least preferred.
- `\toks\langle thread \rangle` holds the submatch information for the $\langle thread \rangle$, as the contents of a property list.
- `\g__regex_charcode_intarray` and `\g__regex_catcode_intarray` hold the character codes and category codes of tokens at each $\langle position \rangle$ in the query.
- `\g__regex_balance_intarray` holds the balance of begin-group and end-group character tokens which appear before that point in the token list.
- `\toks\langle position \rangle` holds $\langle tokens \rangle$ which o- and x-expand to the $\langle position \rangle$ -th token in the query.
- `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray` and `\g__regex_submatch_end_intarray` hold, for each submatch (as would be extracted by `\regex_extract_all:nnN`), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice max_state , and the used registers go up to `\l__regex_submatch_int`. They are organized in blocks of `\l__regex_capturing_group_int` entries, each block corresponding to one match with all its submatches stored in consecutive entries.

`\count` registers are not abused, which means that we can safely use named integers in this module. Note that `\box` registers are not abused either; maybe we could leverage those for some purpose.

The code is structured as follows. Variables are introduced in the relevant section. First we present some generic helper functions. Then comes the code for compiling a regular expression, and for showing the result of the compilation. The building phase converts a compiled regex to NFA states, and the automaton is run by the code in the following section. The only remaining brick is parsing the replacement text and performing the replacement. We are then ready for all the user functions. Finally, messages, and a little bit of tracing code.

37.2 Helpers

`__regex_int_eval:w` Access the primitive: performance is key here, so we do not use the slower route *via* `\int_eval:n`.

```

18506 \cs_new_eq:NN \__regex_int_eval:w \etex_numexpr:D
18507 % \end{macrocode}
18508 % \end{macro}
18509 %
18510 % \begin{macro}{\__regex_standard_escapechar:}
18511 % Make the \tn{escapechar} into the standard backslash.
18512 % \begin{macrocode}
18513 \cs_new_protected:Npn \__regex_standard_escapechar:
18514 { \int_set:Nn \tex_escapechar:D { '\ } }
```

(End definition for `__regex_int_eval:w`.)

`__regex_toks_use:w` Unpack a `\toks` given its number.

```

18515 \cs_new:Npn \__regex_toks_use:w { \tex_the:D \tex_toks:D }

```

(End definition for __regex_toks_use:w.)

`__regex_toks_clear:N` Empty a `\toks` or set it to a value, given its number.

```

\__regex_toks_set:Nn
\__regex_toks_set:No
18516 \cs_new_protected:Npn \__regex_toks_clear:N #1
18517 { \__regex_toks_set:Nn #1 { } }
18518 \cs_new_eq:NN \__regex_toks_set:Nn \tex_toks:D
18519 \cs_new_protected:Npn \__regex_toks_set:No #1
18520 { \__regex_toks_set:Nn #1 \exp_after:wN }

```

(End definition for __regex_toks_clear:N and __regex_toks_set:Nn.)

`__regex_toks_memcpy:NNn` Copy #3 `\toks` registers from #2 onwards to #1 onwards, like C's `memcpy`.

```

18521 \cs_new_protected:Npn \__regex_toks_memcpy:NNn #1#2#3
18522 {
18523   \prg_replicate:nn {#3}
18524   {
18525     \tex_toks:D #1 = \tex_toks:D #2
18526     \int_incr:N #1
18527     \int_incr:N #2
18528   }
18529 }

```

(End definition for __regex_toks_memcpy:NNn.)

`__regex_toks_put_left:Nx` During the building phase we wish to add x-expanded material to `\toks`, either to the left

`__regex_toks_put_right:Nx` or to the right. The expansion is done “by hand” for optimization (these operations are

`__regex_toks_put_right:Nn` used quite a lot). The `Nn` version of `__regex_toks_put_right:Nx` is provided because

it is more efficient than x-expanding with `\exp_not:n`.

```

18530 \cs_new_protected:Npn \__regex_toks_put_left:Nx #1#2
18531 {
18532   \cs_set:Npx \__regex_tmp:w { #2 }
18533   \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
18534   { \exp_after:wN \__regex_tmp:w \tex_the:D \tex_toks:D #1 }
18535 }
18536 \cs_new_protected:Npn \__regex_toks_put_right:Nx #1#2
18537 {
18538   \cs_set:Npx \__regex_tmp:w {#2}
18539   \tex_toks:D #1 \exp_after:wN
18540   { \tex_the:D \tex_toks:D \exp_after:wN #1 \__regex_tmp:w }
18541 }
18542 \cs_new_protected:Npn \__regex_toks_put_right:Nn #1#2
18543 { \tex_toks:D #1 \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }

```

(End definition for __regex_toks_put_left:Nx and __regex_toks_put_right:Nx.)

`__regex_curr_cs_to_str:` Expands to the string representation of the token (known to be a control sequence) at the current position `\l__regex_curr_pos_int`. It should only be used in x-expansion to avoid losing a leading space.

```

18544 \cs_new:Npn \__regex_curr_cs_to_str:
18545 {
18546   \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
18547   \tex_the:D \tex_toks:D \l__regex_curr_pos_int
18548 }

```

(End definition for `_regex_curr_cs_to_str:.`)

37.2.1 Constants and variables

`_regex_tmp:w` Temporary function used for various short-term purposes.

```
18549 \cs_new:Npn \_regex_tmp:w { }
```

(End definition for `_regex_tmp:w`.)

`\l__regex_internal_a_tl` Temporary variables used for various purposes.

```
\l__regex_internal_b_tl
\l__regex_internal_a_int
\l__regex_internal_b_int
\l__regex_internal_c_int
\l__regex_internal_bool
\l__regex_internal_seq
\g__regex_internal_tl
18550 \tl_new:N \l__regex_internal_a_tl
18551 \tl_new:N \l__regex_internal_b_tl
18552 \int_new:N \l__regex_internal_a_int
18553 \int_new:N \l__regex_internal_b_int
18554 \int_new:N \l__regex_internal_c_int
18555 \bool_new:N \l__regex_internal_bool
18556 \seq_new:N \l__regex_internal_seq
18557 \tl_new:N \g__regex_internal_tl
```

(End definition for `\l__regex_internal_a_tl` and others.)

`\l__regex_build_tl` This temporary variable is specifically for use with the `tl_build` machinery.

```
18558 \tl_new:N \l__regex_build_tl
```

(End definition for `\l__regex_build_tl`.)

`\c__regex_no_match_regex` This regular expression matches nothing, but is still a valid regular expression. We could use a failing assertion, but I went for an empty class. It is used as the initial value for regular expressions declared using `\regex_new:N`.

```
18559 \tl_const:Nn \c__regex_no_match_regex
18560 {
18561   \__regex_branch:n
18562   { \__regex_class:NnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
18563 }
```

(End definition for `\c__regex_no_match_regex`.)

`\g__regex_charcode_intarray`
`\g__regex_catcode_intarray`
`\g__regex_balance_intarray` The first thing we do when matching is to go once through the query token list and store the information for each token into `\g__regex_charcode_intarray`, `\g__regex_catcode_intarray` and `\toks` registers. We also store the balance of begin-group/end-group characters into `\g__regex_balance_intarray`.

```
18564 \intarray_new:Nn \g__regex_charcode_intarray { 65536 }
18565 \intarray_new:Nn \g__regex_catcode_intarray { 65536 }
18566 \intarray_new:Nn \g__regex_balance_intarray { 65536 }
```

(End definition for `\g__regex_charcode_intarray`, `\g__regex_catcode_intarray`, and `\g__regex_balance_intarray`.)

`\l__regex_balance_int` During this phase, `\l__regex_balance_int` counts the balance of begin-group and end-group character tokens which appear before a given point in the token list. This variable is also used to keep track of the balance in the replacement text.

```
18567 \int_new:N \l__regex_balance_int
```

(End definition for `\l__regex_balance_int`.)

`\l__regex_cs_name_tl` This variable is used in `__regex_item_cs:n` to store the csname of the currently-tested token when the regex contains a sub-regex for testing csnames.

```
18568 \tl_new:N \l__regex_cs_name_tl
```

(End definition for `\l__regex_cs_name_tl`.)

37.2.2 Testing characters

`\c__regex_ascii_min_int`

`\c__regex_ascii_max_control_int`

`\c__regex_ascii_max_int`

```
18569 \int_const:Nn \c__regex_ascii_min_int { 0 }
```

```
18570 \int_const:Nn \c__regex_ascii_max_control_int { 31 }
```

```
18571 \int_const:Nn \c__regex_ascii_max_int { 127 }
```

(End definition for `\c__regex_ascii_min_int`, `\c__regex_ascii_max_control_int`, and `\c__regex_ascii_max_int`.)

`\c__regex_ascii_lower_int`

```
18572 \int_const:Nn \c__regex_ascii_lower_int { 'a' - 'A' }
```

(End definition for `\c__regex_ascii_lower_int`.)

`__regex_break_point:TF`

`__regex_break_true:w`

When testing whether a character of the query token list matches a given character class in the regular expression, we often have to test it against several ranges of characters, checking if any one of those matches. This is done with a structure like

```
<test1> ... <testn>
__regex_break_point:TF {<true code>} {<false code>}
```

If any of the tests succeeds, it calls `__regex_break_true:w`, which cleans up and leaves `<true code>` in the input stream. Otherwise, `__regex_break_point:TF` leaves the `<false code>` in the input stream.

```
18573 \cs_new_protected:Npn \__regex_break_true:w
```

```
18574   #1 \__regex_break_point:TF #2 #3 {#2}
```

```
18575 \cs_new_protected:Npn \__regex_break_point:TF #1 #2 { #2 }
```

(End definition for `__regex_break_point:TF` and `__regex_break_true:w`.)

`__regex_item_reverse:n`

This function makes showing regular expressions easier, and lets us define `\D` in terms of `\d` for instance. There is a subtlety: the end of the query is marked by `-2`, and thus matches `\D` and other negated properties; this case is caught by another part of the code.

```
18576 \cs_new_protected:Npn \__regex_item_reverse:n #1
```

```
18577   {
```

```
18578     #1
```

```
18579     \__regex_break_point:TF { } \__regex_break_true:w
```

```
18580   }
```

(End definition for `__regex_item_reverse:n`.)

`__regex_item_caseful_equal:n`

`__regex_item_caseful_range:nn`

Simple comparisons triggering `__regex_break_true:w` when true.

```
18581 \cs_new_protected:Npn \__regex_item_caseful_equal:n #1
```

```
18582   {
```

```
18583     \if_int_compare:w #1 = \l__regex_curr_char_int
```

```
18584     \exp_after:wN \__regex_break_true:w
```

```
18585     \fi:
```

```
18586   }
```

```

18587 \cs_new_protected:Npn \__regex_item_caseful_range:nn #1 #2
18588 {
18589   \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
18590   \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
18591   \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
18592   \fi:
18593   \fi:
18594 }

```

(End definition for __regex_item_caseful_equal:n and __regex_item_caseful_range:nn.)

__regex_item_caseless_equal:n For caseless matching, we perform the test both on the `current_char` and on the `case_changed_char`. Before doing the second set of tests, we make sure that `case_changed_char` has been computed.

```

18595 \cs_new_protected:Npn \__regex_item_caseless_equal:n #1
18596 {
18597   \if_int_compare:w #1 = \l__regex_curr_char_int
18598   \exp_after:wN \__regex_break_true:w
18599   \fi:
18600   \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
18601   \__regex_compute_case_changed_char:
18602   \fi:
18603   \if_int_compare:w #1 = \l__regex_case_changed_char_int
18604   \exp_after:wN \__regex_break_true:w
18605   \fi:
18606 }
18607 \cs_new_protected:Npn \__regex_item_caseless_range:nn #1 #2
18608 {
18609   \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
18610   \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
18611   \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
18612   \fi:
18613   \fi:
18614   \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
18615   \__regex_compute_case_changed_char:
18616   \fi:
18617   \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
18618   \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
18619   \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
18620   \fi:
18621   \fi:
18622 }

```

(End definition for __regex_item_caseless_equal:n and __regex_item_caseless_range:nn.)

__regex_compute_case_changed_char: This function is called when `\l__regex_case_changed_char_int` has not yet been computed (or rather, when it is set to the marker value `\c_max_int`). If the current character code is in the range [65, 90] (upper-case), then add 32, making it lowercase. If it is in the lower-case letter range [97, 122], subtract 32.

```

18623 \cs_new_protected:Npn \__regex_compute_case_changed_char:
18624 {
18625   \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int
18626   \if_int_compare:w \l__regex_curr_char_int > 'Z \exp_stop_f:
18627   \if_int_compare:w \l__regex_curr_char_int > 'z \exp_stop_f: \else:

```

```

18628         \if_int_compare:w \l__regex_curr_char_int < 'a \exp_stop_f: \else:
18629         \int_sub:Nn \l__regex_case_changed_char_int
18630         { \c__regex_ascii_lower_int }
18631     \fi:
18632 \fi:
18633 \else:
18634     \if_int_compare:w \l__regex_curr_char_int < 'A \exp_stop_f: \else:
18635     \int_add:Nn \l__regex_case_changed_char_int
18636     { \c__regex_ascii_lower_int }
18637 \fi:
18638 \fi:
18639 }

```

(End definition for `__regex_compute_case_changed_char:.`)

`__regex_item_equal:n` Those must always be defined to expand to a `caseful` (default) or `caseless` version, and not be protected: they must expand when compiling, to hard-code which tests are caseless or caseful.

```

18640 \cs_new_eq:NN \__regex_item_equal:n ?
18641 \cs_new_eq:NN \__regex_item_range:nn ?

```

(End definition for `__regex_item_equal:n` and `__regex_item_range:nn`.)

`__regex_item_catcode:nT` The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

```

18642 \cs_new_protected:Npn \__regex_item_catcode:
18643 {
18644     "
18645     \if_case:w \l__regex_curr_catcode_int
18646         1      \or: 4      \or: 10      \or: 40
18647     \or: 100   \or:      \or: 1000   \or: 4000
18648     \or: 10000 \or:      \or: 100000 \or: 400000
18649     \or: 1000000 \or: 4000000 \else: 1*0
18650     \fi:
18651 }
18652 \cs_new_protected:Npn \__regex_item_catcode:nT #1
18653 {
18654     \if_int_odd:w \int_eval:n { #1 / \__regex_item_catcode: } \exp_stop_f:
18655     \exp_after:wN \use:n
18656     \else:
18657     \exp_after:wN \use_none:n
18658     \fi:
18659 }
18660 \cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
18661 { \__regex_item_catcode:nT {#1} { \__regex_item_reverse:n {#2} } }

```

(End definition for `__regex_item_catcode:nT`, `__regex_item_catcode_reverse:nT`, and `__regex_item_catcode:.`)

`__regex_item_exact:nn` This matches an exact `<category>-<character code>` pair, or an exact control sequence, more precisely one of several possible control sequences.

```

18662 \cs_new_protected:Npn \__regex_item_exact:nn #1#2

```

```

18663 {
18664     \if_int_compare:w #1 = \l__regex_curr_catcode_int
18665     \if_int_compare:w #2 = \l__regex_curr_char_int
18666     \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
18667     \fi:
18668     \fi:
18669 }
18670 \cs_new_protected:Npn \__regex_item_exact_cs:n #1
18671 {
18672     \int_compare:nNnTF \l__regex_curr_catcode_int = 0
18673     {
18674         \tl_set:Nx \l__regex_internal_a_tl
18675         { \scan_stop: \__regex_curr_cs_to_str: \scan_stop: }
18676         \tl_if_in:noTF { \scan_stop: #1 \scan_stop: }
18677         \l__regex_internal_a_tl
18678         { \__regex_break_true:w } { }
18679     }
18680     { }
18681 }

```

(End definition for `__regex_item_exact:nn` and `__regex_item_exact_cs:n`.)

`__regex_item_cs:n` Match a control sequence (the argument is a compiled regex). First test the catcode of the current token to be zero. Then perform the matching test, and break if the csname indeed matches. The three `\exp_after:wN` expand the contents of the `\toks<current position>` (of the form `\exp_not:n {<control sequence>}`) to `<control sequence>`. We store the cs name before building states for the cs, as those states may overlap with toks registers storing the user's input.

```

18682 \cs_new_protected:Npn \__regex_item_cs:n #1
18683 {
18684     \int_compare:nNnTF \l__regex_curr_catcode_int = 0
18685     {
18686         \group_begin:
18687         \tl_set:Nx \l__regex_cs_name_tl { \__regex_curr_cs_to_str: }
18688         \__regex_single_match:
18689         \__regex_disable_submatches:
18690         \__regex_build_for_cs:n {#1}
18691         \bool_set_eq:NN \l__regex_saved_success_bool
18692         \g__regex_success_bool
18693         \exp_args:NV \__regex_match:n \l__regex_cs_name_tl
18694         \if_meaning:w \c_true_bool \g__regex_success_bool
18695         \group_insert_after:N \__regex_break_true:w
18696         \fi:
18697         \bool_gset_eq:NN \g__regex_success_bool
18698         \l__regex_saved_success_bool
18699         \group_end:
18700     }
18701 }

```

(End definition for `__regex_item_cs:n`.)

37.2.3 Character property tests

`__regex_prop_d:` Character property tests for `\d`, `\W`, *etc.* These character properties are not affected by the `(?i)` option. The characters recognized by each one are as follows: `\d=[0-9]`,
`__regex_prop_h:`
`__regex_prop_s:`
`__regex_prop_v:`
`__regex_prop_w:`
`__regex_prop_N:`

`\w=[0-9A-Z_a-z]`, `\s=[_\^\^I\^J\^L\^M]`, `\h=[_\^\^I]`, `\v=[\^J-\^M]`, and the upper case counterparts match anything that the lower case does not match. The order in which the various tests appear is optimized for usual mostly lower case letter text.

```

18702 \cs_new_protected:Npn \__regex_prop_d:
18703 { \__regex_item_caseful_range:nn { '0 } { '9 } }
18704 \cs_new_protected:Npn \__regex_prop_h:
18705 {
18706   \__regex_item_caseful_equal:n { '\ }
18707   \__regex_item_caseful_equal:n { '\^I }
18708 }
18709 \cs_new_protected:Npn \__regex_prop_s:
18710 {
18711   \__regex_item_caseful_equal:n { '\ }
18712   \__regex_item_caseful_equal:n { '\^I }
18713   \__regex_item_caseful_equal:n { '\^J }
18714   \__regex_item_caseful_equal:n { '\^L }
18715   \__regex_item_caseful_equal:n { '\^M }
18716 }
18717 \cs_new_protected:Npn \__regex_prop_v:
18718 { \__regex_item_caseful_range:nn { '\^J } { '\^M } } % lf, vtab, ff, cr
18719 \cs_new_protected:Npn \__regex_prop_w:
18720 {
18721   \__regex_item_caseful_range:nn { 'a } { 'z }
18722   \__regex_item_caseful_range:nn { 'A } { 'Z }
18723   \__regex_item_caseful_range:nn { '0 } { '9 }
18724   \__regex_item_caseful_equal:n { '_' }
18725 }
18726 \cs_new_protected:Npn \__regex_prop_N:
18727 {
18728   \__regex_item_reverse:n
18729   { \__regex_item_caseful_equal:n { '\^J } }
18730 }

```

(End definition for __regex_prop_d: and others.)

```

\__regex_posix_alnum: POSIX properties. No surprise.
\__regex_posix_alpha: 18731 \cs_new_protected:Npn \__regex_posix_alnum:
\__regex_posix_ascii: 18732 { \__regex_posix_alpha: \__regex_posix_digit: }
\__regex_posix_blank: 18733 \cs_new_protected:Npn \__regex_posix_alpha:
\__regex_posix_cntrl: 18734 { \__regex_posix_lower: \__regex_posix_upper: }
\__regex_posix_digit: 18735 \cs_new_protected:Npn \__regex_posix_ascii:
\__regex_posix_graph: 18736 {
\__regex_posix_lower: 18737   \__regex_item_caseful_range:nn
\__regex_posix_print: 18738   \c__regex_ascii_min_int
\__regex_posix_punct: 18739   \c__regex_ascii_max_int
\__regex_posix_space: 18740 }
\__regex_posix_upper: 18741 \cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\__regex_posix_word: 18742 \cs_new_protected:Npn \__regex_posix_cntrl:
\__regex_posix_xdigit: 18743 {
18744   \__regex_item_caseful_range:nn
18745   \c__regex_ascii_min_int
18746   \c__regex_ascii_max_control_int
18747   \__regex_item_caseful_equal:n \c__regex_ascii_max_int
18748 }

```

```

18749 \cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
18750 \cs_new_protected:Npn \__regex_posix_graph:
18751   { \__regex_item_caseful_range:nn { '!' } { '~ } }
18752 \cs_new_protected:Npn \__regex_posix_lower:
18753   { \__regex_item_caseful_range:nn { 'a' } { 'z' } }
18754 \cs_new_protected:Npn \__regex_posix_print:
18755   { \__regex_item_caseful_range:nn { '\' } { '~ } }
18756 \cs_new_protected:Npn \__regex_posix_punct:
18757   {
18758     \__regex_item_caseful_range:nn { '!' } { '/' }
18759     \__regex_item_caseful_range:nn { ':' } { '@' }
18760     \__regex_item_caseful_range:nn { '[' } { '"' }
18761     \__regex_item_caseful_range:nn { '\{ } { '~ }
18762   }
18763 \cs_new_protected:Npn \__regex_posix_space:
18764   {
18765     \__regex_item_caseful_equal:n { '\' }
18766     \__regex_item_caseful_range:nn { '^I' } { '^M' }
18767   }
18768 \cs_new_protected:Npn \__regex_posix_upper:
18769   { \__regex_item_caseful_range:nn { 'A' } { 'Z' } }
18770 \cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
18771 \cs_new_protected:Npn \__regex_posix_xdigit:
18772   {
18773     \__regex_posix_digit:
18774     \__regex_item_caseful_range:nn { 'A' } { 'F' }
18775     \__regex_item_caseful_range:nn { 'a' } { 'f' }
18776   }

```

(End definition for `__regex_posix_alnum:` and others.)

37.2.4 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting `\n` to the character 10, *etc.* In this pass, we also convert any special character (`*`, `?`, `{`, *etc.*) or escaped alphanumeric character into a marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were “raw” characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: `__regex_escape_use:nnnn` *<inline 1>* *<inline 2>* *<inline 3>* *{<token list>}*
The *<token list>* is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function *<inline 1>*, and escaped characters are fed to the function *<inline 2>* within an `x`-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized, and those are replaced by the corresponding character, then fed to *<inline 3>*. The result is then left in the input stream. Spaces are ignored unless escaped.

The conversion is done within an `x`-expanding assignment.

`__regex_escape_use:nnnn` The result is built in `\l__regex_internal_a_tl`, which is then left in the input stream. Tracing code is added as appropriate inside this token list. Go through `#4` once, applying `#1`, `#2`, or `#3` as relevant to each character (after de-escaping it).

```

18777 \__kernel_patch:nnNn
18778 {
18779   \__regex_trace_push:nnN { regex } { 1 } \__regex_escape_use:nnnn
18780   \group_begin:
18781     \tl_set:Nx \l__regex_internal_a_tl
18782     { \__regex_trace_pop:nnN { regex } { 1 } \__regex_escape_use:nnnn }
18783     \use_none:nnn
18784   }
18785 { }
18786 \cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
18787 {
18788   \group_begin:
18789     \tl_clear:N \l__regex_internal_a_tl
18790     \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
18791     \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
18792     \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
18793     \__regex_standard_escapechar:
18794     \tl_gset:Nx \g__regex_internal_tl
18795     { \__kernel_str_to_other_fast:n {#4} }
18796     \tl_put_right:Nx \l__regex_internal_a_tl
18797     {
18798       \exp_after:wN \__regex_escape_loop:N \g__regex_internal_tl
18799       { break } \prg_break_point:
18800     }
18801     \exp_after:wN
18802     \group_end:
18803     \l__regex_internal_a_tl
18804   }

```

(End definition for __regex_escape_use:nnnn.)

__regex_escape_loop:N __regex_escape_loop:N reads one character: if it is special (space, backslash, or end-marker), perform the associated action, otherwise it is simply an unescaped character. After a backslash, the same is done, but unknown characters are “escaped”.

```

18805 \cs_new:Npn \__regex_escape_loop:N #1
18806 {
18807   \cs_if_exist_use:cF { __regex_escape\_token_to_str:N #1:w }
18808   { \__regex_escape_unescaped:N #1 }
18809   \__regex_escape_loop:N
18810 }
18811 \cs_new:cpn { __regex_escape\_c_backslash_str :w }
18812 \__regex_escape_loop:N #1
18813 {
18814   \cs_if_exist_use:cF { __regex_escape\_token_to_str:N #1:w }
18815   { \__regex_escape_escaped:N #1 }
18816   \__regex_escape_loop:N
18817 }

```

(End definition for __regex_escape_loop:N and __regex_escape_w.)

__regex_escape_unescaped:N Those functions are never called before being given a new meaning, so their definitions here don’t matter.

```

\__regex_escape_escaped:N
\__regex_escape_raw:N
18818 \cs_new_eq:NN \__regex_escape_unescaped:N ?
18819 \cs_new_eq:NN \__regex_escape_escaped:N ?
18820 \cs_new_eq:NN \__regex_escape_raw:N ?

```

(End definition for `__regex_escape_unescaped:N`, `__regex_escape_escaped:N`, and `__regex_escape_raw:N`.)

`__regex_escape_break:w` The loop is ended upon seeing the end-marker “break”, with an error if the string ended in a backslash. Spaces are ignored, and `\a`, `\e`, `\f`, `\n`, `\r`, `\t` take their meaning here.

`__regex_escape_/break:w`

`__regex_escape_/a:w` 18821 `\cs_new_eq:NN __regex_escape_break:w \prg_break:`

`__regex_escape_/e:w` 18822 `\cs_new:cpn { __regex_escape_/break:w }`

`__regex_escape_/f:w` 18823 `{`

`__regex_escape_/n:w` 18824 `__kernel_msg_expandable_error:nn { kernel } { trailing-backslash }`

`__regex_escape_/r:w` 18825 `\prg_break:`

`__regex_escape_/t:w` 18826 `}`

`__regex_escape_□:w` 18827 `\cs_new:cpn { __regex_escape_~:w } { }`

18828 `\cs_new:cpx { __regex_escape_/a:w }`

18829 `{ \exp_not:N __regex_escape_raw:N \iow_char:N ^^G }`

18830 `\cs_new:cpx { __regex_escape_/t:w }`

18831 `{ \exp_not:N __regex_escape_raw:N \iow_char:N ^^I }`

18832 `\cs_new:cpx { __regex_escape_/n:w }`

18833 `{ \exp_not:N __regex_escape_raw:N \iow_char:N ^^J }`

18834 `\cs_new:cpx { __regex_escape_/f:w }`

18835 `{ \exp_not:N __regex_escape_raw:N \iow_char:N ^^L }`

18836 `\cs_new:cpx { __regex_escape_/r:w }`

18837 `{ \exp_not:N __regex_escape_raw:N \iow_char:N ^^M }`

18838 `\cs_new:cpx { __regex_escape_/e:w }`

18839 `{ \exp_not:N __regex_escape_raw:N \iow_char:N ^^[}`

(End definition for `__regex_escape_break:w` and others.)

`__regex_escape_/x:w` When `\x` is encountered, `__regex_escape_x_test:N` is responsible for grabbing some hexadecimal digits, and feeding the result to `__regex_escape_x_end:w`. If the number is too big interrupt the assignment and produce an error, otherwise call `__regex_escape_raw:N` on the corresponding character token.

`__regex_escape_x_end:w`

`__regex_escape_x_large:n`

18840 `\cs_new:cpn { __regex_escape_/x:w } __regex_escape_loop:N`

18841 `{`

18842 `\exp_after:wN __regex_escape_x_end:w`

18843 `\int_value:w "0 __regex_escape_x_test:N`

18844 `}`

18845 `\cs_new:Npn __regex_escape_x_end:w #1 ;`

18846 `{`

18847 `\int_compare:nNnTF {#1} > \c_max_char_int`

18848 `{`

18849 `__kernel_msg_expandable_error:nnff { kernel } { x-overflow }`

18850 `{#1} { \int_to_Hex:n {#1} }`

18851 `}`

18852 `{`

18853 `\exp_last_unbraced:Nf __regex_escape_raw:N`

18854 `{ \char_generate:nn {#1} { 12 } }`

18855 `}`

18856 `}`

(End definition for `__regex_escape_/x:w`, `__regex_escape_x_end:w`, and `__regex_escape_x_large:n`.)

`__regex_escape_x_test:N` Find out whether the first character is a left brace (allowing any number of hexadecimal digits), or not (allowing up to two hexadecimal digits). We need to check for the end-of-string marker. Eventually, call either `__regex_escape_x_loop:N` or `__regex_escape_x:N`.

`__regex_escape_x_testii:N`

```

18857 \cs_new:Npn \__regex_escape_x_test:N #1
18858 {
18859   \str_if_eq_x:nnTF {#1} { break } { ; }
18860   {
18861     \if_charcode:w \c_space_token #1
18862       \exp_after:wN \__regex_escape_x_test:N
18863     \else:
18864       \exp_after:wN \__regex_escape_x_testii:N
18865       \exp_after:wN #1
18866     \fi:
18867   }
18868 }
18869 \cs_new:Npn \__regex_escape_x_testii:N #1
18870 {
18871   \if_charcode:w \c_left_brace_str #1
18872     \exp_after:wN \__regex_escape_x_loop:N
18873   \else:
18874     \__regex_hexadecimal_use:NTF #1
18875     { \exp_after:wN \__regex_escape_x:N }
18876     { ; \exp_after:wN \__regex_escape_loop:N \exp_after:wN #1 }
18877   \fi:
18878 }

```

(End definition for __regex_escape_x_test:N and __regex_escape_x_testii:N.)

__regex_escape_x:N This looks for the second digit in the unbraced case.

```

18879 \cs_new:Npn \__regex_escape_x:N #1
18880 {
18881   \str_if_eq_x:nnTF {#1} { break } { ; }
18882   {
18883     \__regex_hexadecimal_use:NTF #1
18884     { ; \__regex_escape_loop:N }
18885     { ; \__regex_escape_loop:N #1 }
18886   }
18887 }

```

(End definition for __regex_escape_x:N.)

__regex_escape_x_loop:N Grab hexadecimal digits, skip spaces, and at the end, check that there is a right brace, otherwise raise an error outside the assignment.

```

18888 \cs_new:Npn \__regex_escape_x_loop:N #1
18889 {
18890   \str_if_eq_x:nnTF {#1} { break }
18891   { ; \__regex_escape_x_loop_error:n { } {#1} }
18892   {
18893     \__regex_hexadecimal_use:NTF #1
18894     { \__regex_escape_x_loop:N }
18895     {
18896       \token_if_eq_charcode:NNTF \c_space_token #1
18897       { \__regex_escape_x_loop:N }
18898       {
18899         ;
18900         \exp_after:wN
18901         \token_if_eq_charcode:NNTF \c_right_brace_str #1

```

```

18902             { \_regex_escape_loop:N }
18903             { \_regex_escape_x_loop_error:n {#1} }
18904         }
18905     }
18906 }
18907 }
18908 \cs_new:Npn \_regex_escape_x_loop_error:n #1
18909 {
18910     \_kernel_msg_expandable_error:nnn { kernel } { x-missing-rbrace } {#1}
18911     \_regex_escape_loop:N #1
18912 }

```

(End definition for _regex_escape_x_loop:N and _regex_escape_x_loop_error:.)

_regex_hexadecimal_use:NTF TeX detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

18913 \prg_new_conditional:Npnn \_regex_hexadecimal_use:N #1 { TF }
18914 {
18915     \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
18916     #1 \prg_return_true:
18917     \else:
18918         \if_case:w
18919             \int_eval:n { \exp_after:wN ‘ \token_to_str:N #1 - ‘a }
18920             A
18921             \or: B
18922             \or: C
18923             \or: D
18924             \or: E
18925             \or: F
18926             \else:
18927                 \prg_return_false:
18928                 \exp_after:wN \use_none:n
18929             \fi:
18930             \prg_return_true:
18931         \fi:
18932     }

```

(End definition for _regex_hexadecimal_use:NTF.)

_regex_char_if_alphanumeric:NTF These two tests are used in the first pass when parsing a regular expression. That pass is responsible for finding escaped and non-escaped characters, and recognizing which ones have special meanings and which should be interpreted as “raw” characters. Namely,

- alphanumeric are “raw” if they are not escaped, and may have a special meaning when escaped;
- non-alphanumeric printable ascii characters are “raw” if they are escaped, and may have a special meaning when not escaped;
- characters other than printable ascii are always “raw”.

The code is ugly, and highly based on magic numbers and the ascii codes of characters. This is mostly unavoidable for performance reasons. Maybe the tests can be optimized a little bit more. Here, “alphanumeric” means 0–9, A–Z, a–z; “special” character means non-alphanumeric but printable ascii, from space (hex 20) to del (hex 7E).

```

18933 \prg_new_conditional:Npnn \__regex_char_if_special:N #1 { TF }
18934 {
18935   \if_int_compare:w '#1 > 'Z \exp_stop_f:
18936   \if_int_compare:w '#1 > 'z \exp_stop_f:
18937   \if_int_compare:w '#1 < \c__regex_ascii_max_int
18938   \prg_return_true: \else: \prg_return_false: \fi:
18939   \else:
18940   \if_int_compare:w '#1 < 'a \exp_stop_f:
18941   \prg_return_true: \else: \prg_return_false: \fi:
18942   \fi:
18943   \else:
18944   \if_int_compare:w '#1 > '9 \exp_stop_f:
18945   \if_int_compare:w '#1 < 'A \exp_stop_f:
18946   \prg_return_true: \else: \prg_return_false: \fi:
18947   \else:
18948   \if_int_compare:w '#1 < '0 \exp_stop_f:
18949   \if_int_compare:w '#1 < '\' \exp_stop_f:
18950   \prg_return_false: \else: \prg_return_true: \fi:
18951   \else: \prg_return_false: \fi:
18952   \fi:
18953   \fi:
18954 }
18955 \prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
18956 {
18957   \if_int_compare:w '#1 > 'Z \exp_stop_f:
18958   \if_int_compare:w '#1 > 'z \exp_stop_f:
18959   \prg_return_false:
18960   \else:
18961   \if_int_compare:w '#1 < 'a \exp_stop_f:
18962   \prg_return_false: \else: \prg_return_true: \fi:
18963   \fi:
18964   \else:
18965   \if_int_compare:w '#1 > '9 \exp_stop_f:
18966   \if_int_compare:w '#1 < 'A \exp_stop_f:
18967   \prg_return_false: \else: \prg_return_true: \fi:
18968   \else:
18969   \if_int_compare:w '#1 < '0 \exp_stop_f:
18970   \prg_return_false: \else: \prg_return_true: \fi:
18971   \fi:
18972   \fi:
18973 }

```

(End definition for __regex_char_if_alphanumeric:N and __regex_char_if_special:N.)

37.3 Compiling

A regular expression starts its life as a string of characters. In this section, we convert it to internal instructions, resulting in a “compiled” regular expression. This compiled expression is then turned into states of an automaton in the building phase. Compiled regular expressions consist of the following:

- __regex_class:NnnnN $\langle \text{boolean} \rangle$ $\{ \langle \text{tests} \rangle \}$ $\{ \langle \text{min} \rangle \}$ $\{ \langle \text{more} \rangle \}$ $\langle \text{lazyness} \rangle$

- `__regex_group:nnnN {<branches>} {<min>} {<more>} {<lazyness>}`, also `__regex_group_no_capture:nnnN` and `__regex_group_resetting:nnnN` with the same syntax.
- `__regex_branch:n {<contents>}`
- `__regex_command_K:`
- `__regex_assertion:Nn <boolean> {<assertion test>}`, where the `<assertion test>` is `__regex_b_test:` or `{__regex_anchor:N <integer>}`

Tests can be the following:

- `__regex_item_caseful_equal:n {<char code>}`
- `__regex_item_caseless_equal:n {<char code>}`
- `__regex_item_caseful_range:nn {<min>} {<max>}`
- `__regex_item_caseless_range:nn {<min>} {<max>}`
- `__regex_item_catcode:nT {<catcode bitmap>} {<tests>}`
- `__regex_item_catcode_reverse:nT {<catcode bitmap>} {<tests>}`
- `__regex_item_reverse:n {<tests>}`
- `__regex_item_exact:nn {<catcode>} {<char code>}`
- `__regex_item_exact_cs:n {<csnames>}`, more precisely given as `<csname> \scan_stop: <csname> \scan_stop: <csname>` and so on in a brace group.
- `__regex_item_cs:n {<compiled regex>}`

37.3.1 Variables used when compiling

`\l__regex_group_level_int` We make sure to open the same number of groups as we close.

```
18974 \int_new:N \l__regex_group_level_int
```

(End definition for `\l__regex_group_level_int`.)

`\l__regex_mode_int` While compiling, ten modes are recognized, labelled `-63`, `-23`, `-6`, `-2`, `0`, `2`, `3`, `6`, `23`, `63`.
`\c__regex_cs_in_class_mode_int` See section 37.3.3. We only define some of these as constants.

```
\c__regex_cs_mode_int      18975 \int_new:N \l__regex_mode_int
\c__regex_outer_mode_int   18976 \int_const:Nn \c__regex_cs_in_class_mode_int { -6 }
\c__regex_catcode_mode_int 18977 \int_const:Nn \c__regex_cs_mode_int { -2 }
\c__regex_class_mode_int   18978 \int_const:Nn \c__regex_outer_mode_int { 0 }
\c__regex_catcode_in_class_mode_int 18979 \int_const:Nn \c__regex_catcode_mode_int { 2 }
                             18980 \int_const:Nn \c__regex_class_mode_int { 3 }
                             18981 \int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }
```

(End definition for `\l__regex_mode_int` and others.)

`\l__regex_catcodes_int` We wish to allow constructions such as `\c[^BE](.\cL[a-z]..)`, where the outer catcode test applies to the whole group, but is superseded by the inner catcode test. For this to work, we need to keep track of lists of allowed category codes: `\l__regex_catcodes_int` and `\l__regex_default_catcodes_int` are bitmaps, sums of 4^c , for all allowed catcodes c . The latter is local to each capturing group, and we reset `\l__regex_catcodes_int` to that value after each character or class, changing it only when encountering a `\c` escape. The boolean records whether the list of categories of a catcode test has to be inverted: compare `\c[^BE]` and `\c[BE]`.

```
18982 \int_new:N \l__regex_catcodes_int
18983 \int_new:N \l__regex_default_catcodes_int
18984 \bool_new:N \l__regex_catcodes_bool
```

(End definition for `\l__regex_catcodes_int`, `\l__regex_default_catcodes_int`, and `\l__regex_catcodes_bool`.)

`\c__regex_catcode_C_int` Constants: 4^c for each category, and the sum of all powers of 4.

```
18985 \int_const:Nn \c__regex_catcode_C_int { "1 }
18986 \int_const:Nn \c__regex_catcode_B_int { "4 }
18987 \int_const:Nn \c__regex_catcode_E_int { "10 }
18988 \int_const:Nn \c__regex_catcode_M_int { "40 }
18989 \int_const:Nn \c__regex_catcode_T_int { "100 }
18990 \int_const:Nn \c__regex_catcode_P_int { "1000 }
18991 \int_const:Nn \c__regex_catcode_U_int { "4000 }
18992 \int_const:Nn \c__regex_catcode_D_int { "10000 }
18993 \int_const:Nn \c__regex_catcode_S_int { "100000 }
18994 \int_const:Nn \c__regex_catcode_L_int { "400000 }
18995 \int_const:Nn \c__regex_catcode_O_int { "1000000 }
18996 \int_const:Nn \c__regex_catcode_A_int { "4000000 }
18997 \int_const:Nn \c__regex_all_catcodes_int { "5515155 }
```

(End definition for `\c__regex_catcode_C_int` and others.)

`\l__regex_internal_regex` The compilation step stores its result in this variable.

```
18998 \cs_new_eq:NN \l__regex_internal_regex \c__regex_no_match_regex
```

(End definition for `\l__regex_internal_regex`.)

`\l__regex_show_prefix_seq` This sequence holds the prefix that makes up the line displayed to the user. The various items must be removed from the right, which is tricky with a token list, hence we use a sequence.

```
18999 \seq_new:N \l__regex_show_prefix_seq
```

(End definition for `\l__regex_show_prefix_seq`.)

`\l__regex_show_lines_int` A hack. To know whether a given class has a single item in it or not, we count the number of lines when showing the class.

```
19000 \int_new:N \l__regex_show_lines_int
```

(End definition for `\l__regex_show_lines_int`.)

37.3.2 Generic helpers used when compiling

`__regex_get_digits:NTFw` If followed by some raw digits, collect them one by one in the integer variable #1, and take the true branch. Otherwise, take the false branch.

```

19001 \cs_new_protected:Npn __regex_get_digits:NTFw #1#2#3#4#5
19002 {
19003   __regex_if_raw_digit:NNTF #4 #5
19004   { #1 = #5 __regex_get_digits_loop:nw {#2} }
19005   { #3 #4 #5 }
19006 }
19007 \cs_new:Npn __regex_get_digits_loop:nw #1#2#3
19008 {
19009   __regex_if_raw_digit:NNTF #2 #3
19010   { #3 __regex_get_digits_loop:nw {#1} }
19011   { \scan_stop: #1 #2 #3 }
19012 }
```

(End definition for `__regex_get_digits:NTFw` and `__regex_get_digits_loop:w`.)

`__regex_if_raw_digit:NNTF` Test used when grabbing digits for the {m,n} quantifier. It only accepts non-escaped digits.

```

19013 \prg_new_conditional:Npnn __regex_if_raw_digit:NN #1#2 { TF }
19014 {
19015   \if_meaning:w __regex_compile_raw:N #1
19016   \if_int_compare:w 1 < 1 #2 \exp_stop_f:
19017     \prg_return_true:
19018   \else:
19019     \prg_return_false:
19020   \fi:
19021 \else:
19022   \prg_return_false:
19023 \fi:
19024 }
```

(End definition for `__regex_if_raw_digit:NNTF`.)

37.3.3 Mode

When compiling the NFA corresponding to a given regex string, we can be in ten distinct modes, which we label by some magic numbers:

- 6 `\c{...}` control sequence in a class,
- 2 `\c{...}` control sequence,
- 0 ... outer,
- 2 `\c...` catcode test,
- 6 `\c...` catcode test in a class,
- 63 `\c{[...]}` class inside mode -6,
- 23 `\c{[...]}` class inside mode -2,
- 3 `[...]` class inside mode 0,

23 `\c[...]` class inside mode 2,

63 `[\c[...]]` class inside mode 6.

This list is exhaustive, because `\c` escape sequences cannot be nested, and character classes cannot be nested directly. The choice of numbers is such as to optimize the most useful tests, and make transitions from one mode to another as simple as possible.

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as $m \rightarrow (m - 15)/13$, truncated.
- Grouping, assertion, and anchors are allowed in non-positive even modes (0, -2, -6), and do not change the mode. Otherwise, they trigger an error.
- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from m to $(m - 15)/13$, truncated; also, ranges are recognized.
- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from -2 to 0 or -6 to 3, with error recovery for odd modes.
- Properties (such as the `\d` character class) can appear in any mode.

`_regex_if_in_class:TF` Test whether we are directly in a character class (at the innermost level of nesting). There, many escape sequences are not recognized, and special characters are normal. Also, for every raw character, we must look ahead for a possible raw dash.

```
19025 \cs_new:Npn \_regex_if_in_class:TF
19026 {
19027   \if_int_odd:w \l__regex_mode_int
19028     \exp_after:wN \use_i:nn
19029   \else:
19030     \exp_after:wN \use_ii:nn
19031   \fi:
19032 }
```

(End definition for `_regex_if_in_class:TF`.)

`_regex_if_in_cs:TF` Right braces are special only directly inside control sequences (at the inner-most level of nesting, not counting groups).

```
19033 \cs_new:Npn \_regex_if_in_cs:TF
19034 {
19035   \if_int_odd:w \l__regex_mode_int
19036     \exp_after:wN \use_ii:nn
19037   \else:
19038     \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
19039       \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
19040     \else:
19041       \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
19042     \fi:
19043   \fi:
19044 }
```

(End definition for `_regex_if_in_cs:TF`.)

`_regex_if_in_class_or_catcode:TF` Assertions are only allowed in modes 0, -2, and -6, *i.e.*, even, non-positive modes.

```

19045 \cs_new:Npn \_regex_if_in_class_or_catcode:TF
19046 {
19047   \if_int_odd:w \l__regex_mode_int
19048     \exp_after:wN \use_i:nn
19049   \else:
19050     \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
19051       \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
19052     \else:
19053       \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
19054     \fi:
19055   \fi:
19056 }

```

(End definition for `_regex_if_in_class_or_catcode:TF`.)

`_regex_if_within_catcode:TF` This test takes the true branch if we are in a catcode test, either immediately following it (modes 2 and 6) or in a class on which it applies (modes 23 and 63). This is used to tweak how left brackets behave in modes 2 and 6.

```

19057 \cs_new:Npn \_regex_if_within_catcode:TF
19058 {
19059   \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
19060     \exp_after:wN \use_i:nn
19061   \else:
19062     \exp_after:wN \use_ii:nn
19063   \fi:
19064 }

```

(End definition for `_regex_if_within_catcode:TF`.)

`_regex_chk_c_allowed:T` The `\c` escape sequence is only allowed in modes 0 and 3, *i.e.*, not within any other `\c` escape sequence.

```

19065 \cs_new_protected:Npn \_regex_chk_c_allowed:T
19066 {
19067   \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
19068     \exp_after:wN \use:n
19069   \else:
19070     \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
19071       \exp_after:wN \exp_after:wN \exp_after:wN \use:n
19072     \else:
19073       \__kernel_msg_error:nn { kernel } { c-bad-mode }
19074       \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
19075     \fi:
19076   \fi:
19077 }

```

(End definition for `_regex_chk_c_allowed:T`.)

`_regex_mode_quit_c:` This function changes the mode as it is needed just after a catcode test.

```

19078 \cs_new_protected:Npn \_regex_mode_quit_c:
19079 {
19080   \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int

```

```

19081     \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
19082 \else:
19083     \if_int_compare:w \l__regex_mode_int =
19084         \c__regex_catcode_in_class_mode_int
19085         \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
19086     \fi:
19087 \fi:
19088 }

```

(End definition for `__regex_mode_quit_c:.`)

37.3.4 Framework

`__regex_compile:w` Used when compiling a user regex or a regex for the `\c{...}` escape sequence within another regex. Start building a token list within a group (with `x`-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch. At the end, make sure there are no dangling classes nor groups, close the last branch: we are done building `\l__regex_internal_regex`.

```

19089 \cs_new_protected:Npn \__regex_compile:w
19090 {
19091     \group_begin:
19092     \tl_build_begin:N \l__regex_build_tl
19093     \int_zero:N \l__regex_group_level_int
19094     \int_set_eq:NN \l__regex_default_catcodes_int
19095         \c__regex_all_catcodes_int
19096     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
19097     \cs_set:Npn \__regex_item_equal:n { \__regex_item_caseful_equal:n }
19098     \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
19099     \tl_build_put_right:Nn \l__regex_build_tl
19100         { \__regex_branch:n { \if_false: } \fi: }
19101 }
19102 \cs_new_protected:Npn \__regex_compile_end:
19103 {
19104     \__regex_if_in_class:TF
19105     {
19106         \__kernel_msg_error:nn { kernel } { missing-rbrack }
19107         \use:c { __regex_compile: ]: }
19108         \prg_do_nothing: \prg_do_nothing:
19109     }
19110     { }
19111     \if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f:
19112         \__kernel_msg_error:nnx { kernel } { missing-rparen }
19113         { \int_use:N \l__regex_group_level_int }
19114     \prg_replicate:nn
19115         { \l__regex_group_level_int }
19116     {
19117         \tl_build_put_right:Nn \l__regex_build_tl
19118         {
19119             \if_false: { \fi: }
19120             \if_false: { \fi: } { 1 } { 0 } \c_true_bool
19121         }
19122         \tl_build_end:N \l__regex_build_tl
19123         \exp_args:NNNo
19124         \group_end:

```

```

19125         \tl_build_put_right:Nn \l__regex_build_tl
19126         { \l__regex_build_tl }
19127     }
19128     \fi:
19129     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
19130     \tl_build_end:N \l__regex_build_tl
19131     \exp_args:NNNx
19132     \group_end:
19133     \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
19134 }

```

(End definition for __regex_compile:w and __regex_compile_end:.)

__regex_compile:n The compilation is done between __regex_compile:w and __regex_compile_end:, starting in mode 0. Then __regex_escape_use:nnnn distinguishes special characters, escaped alphanumerics, and raw characters, interpreting \a, \x and other sequences. The 4 trailing \prg_do_nothing: are needed because some functions defined later look up to 4 tokens ahead. Before ending, make sure that any \c{...} is properly closed. No need to check that brackets are closed properly since __regex_compile_end: does that. However, catch the case of a trailing \cL construction.

```

19135 \cs_new_protected:Npn \__regex_compile:n #1
19136 {
19137     \__regex_compile:w
19138     \__regex_standard_escapechar:
19139     \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
19140     \__regex_escape_use:nnnn
19141     {
19142         \__regex_char_if_special:NTF ##1
19143         \__regex_compile_special:N \__regex_compile_raw:N ##1
19144     }
19145     {
19146         \__regex_char_if_alphanumeric:NTF ##1
19147         \__regex_compile_escaped:N \__regex_compile_raw:N ##1
19148     }
19149     { \__regex_compile_raw:N ##1 }
19150     { #1 }
19151     \prg_do_nothing: \prg_do_nothing:
19152     \prg_do_nothing: \prg_do_nothing:
19153     \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
19154     { \__kernel_msg_error:nn { kernel } { c-trailing } }
19155     \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
19156     {
19157         \__kernel_msg_error:nn { kernel } { c-missing-rbrace }
19158         \__regex_compile_end_cs:
19159         \prg_do_nothing: \prg_do_nothing:
19160         \prg_do_nothing: \prg_do_nothing:
19161     }
19162     \__regex_compile_end:
19163 }

```

(End definition for __regex_compile:n.)

__regex_compile_escaped:N If the special character or escaped alphanumeric has a particular meaning in regexes, the corresponding function is used. Otherwise, it is interpreted as a raw character. We

__regex_compile_special:N

distinguish special characters from escaped alphanumeric characters because they behave differently when appearing as an end-point of a range.

```

19164 \cs_new_protected:Npn \__regex_compile_special:N #1
19165 {
19166   \cs_if_exist_use:cF { __regex_compile_#1: }
19167   { \__regex_compile_raw:N #1 }
19168 }
19169 \cs_new_protected:Npn \__regex_compile_escaped:N #1
19170 {
19171   \cs_if_exist_use:cF { __regex_compile_/#1: }
19172   { \__regex_compile_raw:N #1 }
19173 }

```

(End definition for __regex_compile_escaped:N and __regex_compile_special:N.)

`__regex_compile_one:n` This is used after finding one “test”, such as `\d`, or a raw character. If that followed a catcode test (e.g., `\cL`), then restore the mode. If we are not in a class, then the test is “standalone”, and we need to add `__regex_class:NnnnN` and search for quantifiers. In any case, insert the test, possibly together with a catcode test if appropriate.

```

19174 \cs_new_protected:Npn \__regex_compile_one:n #1
19175 {
19176   \__regex_mode_quit_c:
19177   \__regex_if_in_class:TF { }
19178   {
19179     \tl_build_put_right:Nn \l__regex_build_tl
19180     { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
19181   }
19182   \tl_build_put_right:Nx \l__regex_build_tl
19183   {
19184     \if_int_compare:w \l__regex_catcodes_int <
19185     \c__regex_all_catcodes_int
19186     \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
19187     { \exp_not:N \exp_not:n {#1} }
19188     \else:
19189     \exp_not:N \exp_not:n {#1}
19190     \fi:
19191   }
19192   \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
19193   \__regex_if_in_class:TF { } { \__regex_compile_quantifier:w }
19194 }

```

(End definition for __regex_compile_one:n.)

`__regex_compile_abort_tokens:n` This function places the collected tokens back in the input stream, each as a raw character.
`__regex_compile_abort_tokens:x` Spaces are not preserved.

```

19195 \cs_new_protected:Npn \__regex_compile_abort_tokens:n #1
19196 {
19197   \use:x
19198   {
19199     \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
19200     \__regex_compile_raw:N
19201   }
19202 }
19203 \cs_generate_variant:Nn \__regex_compile_abort_tokens:n { x }

```

(End definition for `_regex_compile_abort_tokens:n`.)

37.3.5 Quantifiers

`_regex_compile_quantifier:w` This looks ahead and finds any quantifier (special character equal to either of `?+*{}`).

```

19204 \cs_new_protected:Npn \_regex_compile_quantifier:w #1#2
19205 {
19206   \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
19207   {
19208     \cs_if_exist_use:cF { \_regex_compile_quantifier_#2:w }
19209     { \_regex_compile_quantifier_none: #1 #2 }
19210   }
19211   { \_regex_compile_quantifier_none: #1 #2 }
19212 }

```

(End definition for `_regex_compile_quantifier:w`.)

`_regex_compile_quantifier_none:` Those functions are called whenever there is no quantifier, or a braced construction is invalid (equivalent to no quantifier, and whatever characters were grabbed are left raw).
`_regex_compile_quantifier_abort:xNN`

```

19213 \cs_new_protected:Npn \_regex_compile_quantifier_none:
19214 {
19215   \tl_build_put_right:Nn \l__regex_build_tl
19216   { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
19217 }
19218 \cs_new_protected:Npn \_regex_compile_quantifier_abort:xNN #1#2#3
19219 {
19220   \_regex_compile_quantifier_none:
19221   \_kernel_msg_warning:nxxx { kernel } { invalid-quantifier } {#1} {#3}
19222   \_regex_compile_abort_tokens:x {#1}
19223   #2 #3
19224 }

```

(End definition for `_regex_compile_quantifier_none:` and `_regex_compile_quantifier_abort:xNN`.)

`_regex_compile_quantifier_lazyness:nnNN` Once the “main” quantifier (`?`, `*`, `+` or a braced construction) is found, we check whether it is lazy (followed by a question mark). We then add to the compiled regex a closing brace (ending `_regex_class:NnnnN` and friends), the start-point of the range, its end-point, and a boolean, `true` for lazy and `false` for greedy operators.

```

19225 \cs_new_protected:Npn \_regex_compile_quantifier_lazyness:nnNN #1#2#3#4
19226 {
19227   \str_if_eq:nnTF { #3 #4 } { \_regex_compile_special:N ? }
19228   {
19229     \tl_build_put_right:Nn \l__regex_build_tl
19230     { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
19231   }
19232   {
19233     \tl_build_put_right:Nn \l__regex_build_tl
19234     { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
19235     #3 #4
19236   }
19237 }

```

(End definition for `_regex_compile_quantifier_lazyness:nnNN`.)

For each “basic” quantifier, ?, *, +, feed the correct arguments to `__regex_compile_quantifier_lazyiness:nnNN`, -1 means that there is no upper bound on the number of repetitions.

```

19238 \cs_new_protected:cpn { __regex_compile_quantifier_?:w }
19239 { __regex_compile_quantifier_lazyiness:nnNN { 0 } { 1 } }
19240 \cs_new_protected:cpn { __regex_compile_quantifier_*:w }
19241 { __regex_compile_quantifier_lazyiness:nnNN { 0 } { -1 } }
19242 \cs_new_protected:cpn { __regex_compile_quantifier_+:w }
19243 { __regex_compile_quantifier_lazyiness:nnNN { 1 } { -1 } }

```

(End definition for `__regex_compile_quantifier_?:w`, `__regex_compile_quantifier_*:w`, and `__regex_compile_quantifier_+:w`.)

Three possible syntaxes: $\{\langle int \rangle\}$, $\{\langle int \rangle, \}$, or $\{\langle int \rangle, \langle int \rangle\}$. Any other syntax causes us to abort and put whatever we collected back in the input stream, as raw characters, including the opening brace. Grab a number into `\l__regex_internal_a_int`. If the number is followed by a right brace, the range is $[a, a]$. If followed by a comma, grab one more number, and call the `_ii` or `_iii` auxiliary. Those auxiliaries check for a closing brace, leading to the range $[a, \infty]$ or $[a, b]$, encoded as $\{a\}\{-1\}$ and $\{a\}\{b-a\}$.

```

19244 \cs_new_protected:cpn { __regex_compile_quantifier_ \c_left_brace_str :w }
19245 {
19246   __regex_get_digits:NTFw \l__regex_internal_a_int
19247   { __regex_compile_quantifier_braced_auxi:w }
19248   { __regex_compile_quantifier_abort:xNN { \c_left_brace_str } }
19249 }
19250 \cs_new_protected:Npn __regex_compile_quantifier_braced_auxi:w #1#2
19251 {
19252   \str_case:x:nnF { #1 #2 }
19253   {
19254     { __regex_compile_special:N \c_right_brace_str }
19255     {
19256       \exp_args:No __regex_compile_quantifier_lazyiness:nnNN
19257       { \int_use:N \l__regex_internal_a_int } { 0 }
19258     }
19259     { __regex_compile_special:N , }
19260     {
19261       __regex_get_digits:NTFw \l__regex_internal_b_int
19262       { __regex_compile_quantifier_braced_auxiii:w }
19263       { __regex_compile_quantifier_braced_auxii:w }
19264     }
19265   }
19266   {
19267     __regex_compile_quantifier_abort:xNN
19268     { \c_left_brace_str \int_use:N \l__regex_internal_a_int }
19269     #1 #2
19270   }
19271 }
19272 \cs_new_protected:Npn __regex_compile_quantifier_braced_auxii:w #1#2
19273 {
19274   \str_if_eq:x:nnTF
19275   { #1 #2 } { __regex_compile_special:N \c_right_brace_str }
19276   {
19277     \exp_args:No __regex_compile_quantifier_lazyiness:nnNN
19278     { \int_use:N \l__regex_internal_a_int } { -1 }

```

```

19279     }
19280     {
19281         \_regex_compile_quantifier_abort:xNN
19282         { \c_left_brace_str \int_use:N \l__regex_internal_a_int , }
19283         #1 #2
19284     }
19285 }
19286 \cs_new_protected:Npn \_regex_compile_quantifier_braced_auxiii:w #1#2
19287 {
19288     \str_if_eq_x:nnTF
19289     { #1 #2 } { \_regex_compile_special:N \c_right_brace_str }
19290     {
19291         \if_int_compare:w \l__regex_internal_a_int >
19292         \l__regex_internal_b_int
19293         \_kernel_msg_error:nnxx { kernel } { backwards-quantifier }
19294         { \int_use:N \l__regex_internal_a_int }
19295         { \int_use:N \l__regex_internal_b_int }
19296         \int_zero:N \l__regex_internal_b_int
19297     \else:
19298         \int_sub:Nn \l__regex_internal_b_int \l__regex_internal_a_int
19299     \fi:
19300     \exp_args:Noo \_regex_compile_quantifier_lazyness:nnNN
19301     { \int_use:N \l__regex_internal_a_int }
19302     { \int_use:N \l__regex_internal_b_int }
19303 }
19304 {
19305     \_regex_compile_quantifier_abort:xNN
19306     {
19307         \c_left_brace_str
19308         \int_use:N \l__regex_internal_a_int ,
19309         \int_use:N \l__regex_internal_b_int
19310     }
19311     #1 #2
19312 }
19313 }

```

(End definition for _regex_compile_quantifier_{:w and others.})

37.3.6 Raw characters

_regex_compile_raw_error:N Within character classes, and following catcode tests, some escaped alphanumeric sequences such as \b do not have any meaning. They are replaced by a raw character, after spitting out an error.

```

19314 \cs_new_protected:Npn \_regex_compile_raw_error:N #1
19315 {
19316     \_kernel_msg_error:nnx { kernel } { bad-escape } {#1}
19317     \_regex_compile_raw:N #1
19318 }

```

(End definition for _regex_compile_raw_error:N.)

_regex_compile_raw:N If we are in a character class and the next character is an unescaped dash, this denotes a range. Otherwise, the current character #1 matches itself.

```

19319 \cs_new_protected:Npn \_regex_compile_raw:N #1#2#3

```

```

19320 {
19321   \__regex_if_in_class:TF
19322   {
19323     \str_if_eq:nnTF {#2#3} { \__regex_compile_special:N - }
19324     { \__regex_compile_range:Nw #1 }
19325     {
19326       \__regex_compile_one:n
19327       { \__regex_item_equal:n { \int_value:w '#1 } }
19328       #2 #3
19329     }
19330   }
19331   {
19332     \__regex_compile_one:n
19333     { \__regex_item_equal:n { \int_value:w '#1 } }
19334     #2 #3
19335   }
19336 }

```

(End definition for __regex_compile_raw:N.)

__regex_compile_range:Nw
 __regex_if_end_range:NNTF

We have just read a raw character followed by a dash; this should be followed by an end-point for the range. Valid end-points are: any raw character; any special character, except a right bracket. In particular, escaped characters are forbidden.

```

19337 \prg_new_protected_conditional:Npnn \__regex_if_end_range:NN #1#2 { TF }
19338 {
19339   \if_meaning:w \__regex_compile_raw:N #1
19340   \prg_return_true:
19341   \else:
19342     \if_meaning:w \__regex_compile_special:N #1
19343     \if_charcode:w ] #2
19344     \prg_return_false:
19345     \else:
19346       \prg_return_true:
19347     \fi:
19348     \else:
19349       \prg_return_false:
19350     \fi:
19351   \fi:
19352 }
19353 \cs_new_protected:Npn \__regex_compile_range:Nw #1#2#3
19354 {
19355   \__regex_if_end_range:NNTF #2 #3
19356   {
19357     \if_int_compare:w '#1 > '#3 \exp_stop_f:
19358     \__kernel_msg_error:nxxx { kernel } { range-backwards } {#1} {#3}
19359   \else:
19360     \tl_build_put_right:Nx \l__regex_build_tl
19361     {
19362       \if_int_compare:w '#1 = '#3 \exp_stop_f:
19363       \__regex_item_equal:n
19364       \else:
19365         \__regex_item_range:nn { \int_value:w '#1 }
19366       \fi:
19367       { \int_value:w '#3 }

```

```

19368     }
19369     \fi:
19370 }
19371 {
19372     \_kernel_msg_warning:nxxx { kernel } { range-missing-end }
19373     {#1} { \c_backslash_str #3 }
19374     \tl_build_put_right:Nx \l__regex_build_tl
19375     {
19376         \__regex_item_equal:n { \int_value:w '#1 \exp_stop_f: }
19377         \__regex_item_equal:n { \int_value:w '- \exp_stop_f: }
19378     }
19379     #2#3
19380 }
19381 }

```

(End definition for `__regex_compile_range:Nw` and `__regex_if_end_range:NNTF`.)

37.3.7 Character properties

`__regex_compile_.`: In a class, the dot has no special meaning. Outside, insert `__regex_prop_.`, which matches any character or control sequence, and refuses `-2` (end-marker).

```

19382 \cs_new_protected:cpx { __regex_compile_.: }
19383 {
19384     \exp_not:N \__regex_if_in_class:TF
19385     { \__regex_compile_raw:N . }
19386     { \__regex_compile_one:n \exp_not:c { __regex_prop_.: } }
19387 }
19388 \cs_new_protected:cpn { __regex_prop_.: }
19389 {
19390     \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
19391     \exp_after:wN \__regex_break_true:w
19392     \fi:
19393 }

```

(End definition for `__regex_compile_.` and `__regex_prop_.`.)

`__regex_compile_/d:` The constants `__regex_prop_d:`, *etc.* hold a list of tests which match the corresponding character class, and jump to the `__regex_break_point:TF` marker. As for a normal character, we check for quantifiers.

```

19394 \cs_set_protected:Npn \__regex_tmp:w #1#2
19395 {
19396     \cs_new_protected:cpx { __regex_compile_/#1: }
19397     { \__regex_compile_one:n \exp_not:c { __regex_prop_#1: } }
19398     \cs_new_protected:cpx { __regex_compile_/#2: }
19399     {
19400         \__regex_compile_one:n
19401         { \__regex_item_reverse:n \exp_not:c { __regex_prop_#1: } }
19402     }
19403 }
19404 \__regex_tmp:w d D
19405 \__regex_tmp:w h H
19406 \__regex_tmp:w s S
19407 \__regex_tmp:w v V
19408 \__regex_tmp:w w W

```

```

19409 \cs_new_protected:cpn { __regex_compile_/N: }
19410 { \__regex_compile_one:n \__regex_prop_N: }

```

(End definition for __regex_compile_/d: and others.)

37.3.8 Anchoring and simple assertions

```

\__regex_compile_anchor:NF In modes where assertions are allowed, anchor to the start of the query, the start of the
    \__regex_compile_^: match, or the end of the query, depending on the integer #1. In other modes, #2 treats
    \__regex_compile_/A: the character as raw, with an error for escaped letters ($ is valid in a class, but \A
    \__regex_compile_/G: definitely a mistake on the user's part).
    \__regex_compile_$: 19411 \cs_new_protected:Npn \__regex_compile_anchor:NF #1#2
    \__regex_compile_/Z: 19412 {
    \__regex_compile_/z: 19413 \__regex_if_in_class_or_catcode:TF {#2}
                        19414 {
                        19415 \tl_build_put_right:Nn \l__regex_build_tl
                        19416 { \__regex_assertion:Nn \c_true_bool { \__regex_anchor:N #1 } }
                        19417 }
                        19418 }
19419 \cs_set_protected:Npn \__regex_tmp:w #1#2
19420 {
19421 \cs_new_protected:cpn { __regex_compile_/#1: }
19422 { \__regex_compile_anchor:NF #2 { \__regex_compile_raw_error:N #1 } }
19423 }
19424 \__regex_tmp:w A \l__regex_min_pos_int
19425 \__regex_tmp:w G \l__regex_start_pos_int
19426 \__regex_tmp:w Z \l__regex_max_pos_int
19427 \__regex_tmp:w z \l__regex_max_pos_int
19428 \cs_set_protected:Npn \__regex_tmp:w #1#2
19429 {
19430 \cs_new_protected:cpn { __regex_compile_#1: }
19431 { \__regex_compile_anchor:NF #2 { \__regex_compile_raw:N #1 } }
19432 }
19433 \exp_args:Nx \__regex_tmp:w { \iow_char:N ^ } \l__regex_min_pos_int
19434 \exp_args:Nx \__regex_tmp:w { \iow_char:N $ } \l__regex_max_pos_int

```

(End definition for __regex_compile_anchor:NF and others.)

__regex_compile_/b: Contrarily to ^ and \$, which could be implemented without really knowing what pre-
 __regex_compile_/B: cedes in the token list, this requires more information, namely, the knowledge of the last
 character code.

```

19435 \cs_new_protected:cpn { __regex_compile_/b: }
19436 {
19437 \__regex_if_in_class_or_catcode:TF
19438 { \__regex_compile_raw_error:N b }
19439 {
19440 \tl_build_put_right:Nn \l__regex_build_tl
19441 { \__regex_assertion:Nn \c_true_bool { \__regex_b_test: } }
19442 }
19443 }
19444 \cs_new_protected:cpn { __regex_compile_/B: }
19445 {
19446 \__regex_if_in_class_or_catcode:TF
19447 { \__regex_compile_raw_error:N B }

```

```

19448     {
19449         \tl_build_put_right:Nn \l__regex_build_tl
19450         { \__regex_assertion:Nn \c_false_bool { \__regex_b_test: } }
19451     }
19452 }

```

(End definition for __regex_compile_/b: and __regex_compile_/B:.)

37.3.9 Character classes

`__regex_compile_:` Outside a class, right brackets have no meaning. In a class, change the mode ($m \rightarrow (m - 15)/13$, truncated) to reflect the fact that we are leaving the class. Look for quantifiers, unless we are still in a class after leaving one (the case of `[... \cL[...]...] .` quantifiers).

```

19453 \cs_new_protected:cpn { __regex_compile_[: }
19454 {
19455     \__regex_if_in_class:TF
19456     {
19457         \if_int_compare:w \l__regex_mode_int >
19458             \c__regex_catcode_in_class_mode_int
19459         \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
19460         \fi:
19461         \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
19462         \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
19463         \if_int_odd:w \l__regex_mode_int \else:
19464             \exp_after:wN \__regex_compile_quantifier:w
19465         \fi:
19466     }
19467     { \__regex_compile_raw:N ] }
19468 }

```

(End definition for __regex_compile_[:.)

`__regex_compile_[:` In a class, left brackets might introduce a POSIX character class, or mean nothing. Immediately following `\c<category>`, we must insert the appropriate catcode test, then parse the class; we pre-expand the catcode as an optimization. Otherwise (modes 0, -2 and -6) just parse the class. The mode is updated later.

```

19469 \cs_new_protected:cpn { __regex_compile_[: }
19470 {
19471     \__regex_if_in_class:TF
19472     { \__regex_compile_class_posix_test:w }
19473     {
19474         \__regex_if_within_catcode:TF
19475         {
19476             \exp_after:wN \__regex_compile_class_catcode:w
19477             \int_use:N \l__regex_catcodes_int ;
19478         }
19479         { \__regex_compile_class_normal:w }
19480     }
19481 }

```

(End definition for __regex_compile_[:.)

`__regex_compile_class_normal:w` In the “normal” case, we insert `__regex_class:NnnnN <boolean>` in the compiled code. The `<boolean>` is true for positive classes, and false for negative classes, characterized by

a leading `^`. The auxiliary `_regex_compile_class:TFNN` also checks for a leading `]` which has a special meaning.

```

19482 \cs_new_protected:Npn \_regex_compile_class_normal:w
19483 {
19484   \_regex_compile_class:TFNN
19485   { \_regex_class:NnnnN \c_true_bool }
19486   { \_regex_class:NnnnN \c_false_bool }
19487 }

```

(End definition for `_regex_compile_class_normal:w`.)

`_regex_compile_class_catcode:w` This function is called for a left bracket in modes 2 or 6 (catcode test, and catcode test within a class). In mode 2 the whole construction needs to be put in a class (like single character). Then determine if the class is positive or negative, inserting `_regex_item_catcode:nT` or the `reverse` variant as appropriate, each with the current catcodes bitmap #1 as an argument, and reset the catcodes.

```

19488 \cs_new_protected:Npn \_regex_compile_class_catcode:w #1;
19489 {
19490   \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
19491     \tl_build_put_right:Nn \l__regex_build_tl
19492     { \_regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
19493   \fi:
19494   \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
19495   \_regex_compile_class:TFNN
19496   { \_regex_item_catcode:nT {#1} }
19497   { \_regex_item_catcode_reverse:nT {#1} }
19498 }

```

(End definition for `_regex_compile_class_catcode:w`.)

`_regex_compile_class:TFNN` If the first character is `^`, then the class is negative (use #2), otherwise it is positive (use #1). If the next character is a right bracket, then it should be changed to a raw one.

`_regex_compile_class:NN`

```

19499 \cs_new_protected:Npn \_regex_compile_class:TFNN #1#2#3#4
19500 {
19501   \l__regex_mode_int = \int_value:w \l__regex_mode_int 3 \exp_stop_f:
19502   \str_if_eq:nnTF { #3 #4 } { \_regex_compile_special:N ^ }
19503   {
19504     \tl_build_put_right:Nn \l__regex_build_tl { #2 { \if_false: } \fi: }
19505     \_regex_compile_class:NN
19506   }
19507   {
19508     \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
19509     \_regex_compile_class:NN #3 #4
19510   }
19511 }
19512 \cs_new_protected:Npn \_regex_compile_class:NN #1#2
19513 {
19514   \token_if_eq_charcode:NNTF #2 ]
19515   { \_regex_compile_raw:N #2 }
19516   { #1 #2 }
19517 }

```

(End definition for `_regex_compile_class:TFNN` and `_regex_compile_class:NN`.)

Here we check for a syntax such as [:alpha:]. We also detect [= and [. which have a meaning in POSIX regular expressions, but are not implemented in l3regex. In case we see [:, grab raw characters until hopefully reaching :]. If that's missing, or the POSIX class is unknown, abort. If all is right, add the test to the current class, with an extra `_regex_item_reverse:n` for negative classes.

```

19518 \cs_new_protected:Npn \_regex_compile_class_posix_test:w #1#2
19519 {
19520   \token_if_eq_meaning:NNT \_regex_compile_special:N #1
19521   {
19522     \str_case:nn { #2 }
19523     {
19524       : { \_regex_compile_class_posix:NNNNw }
19525       = {
19526         \_kernel_msg_warning:nx { kernel }
19527         { posix-unsupported } { = }
19528       }
19529       . {
19530         \_kernel_msg_warning:nx { kernel }
19531         { posix-unsupported } { . }
19532       }
19533     }
19534   }
19535   \_regex_compile_raw:N [ #1 #2
19536 }
19537 \cs_new_protected:Npn \_regex_compile_class_posix:NNNNw #1#2#3#4#5#6
19538 {
19539   \str_if_eq:nnTF { #5 #6 } { \_regex_compile_special:N ^ }
19540   {
19541     \bool_set_false:N \l__regex_internal_bool
19542     \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
19543     \_regex_compile_class_posix_loop:w
19544   }
19545   {
19546     \bool_set_true:N \l__regex_internal_bool
19547     \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
19548     \_regex_compile_class_posix_loop:w #5 #6
19549   }
19550 }
19551 \cs_new:Npn \_regex_compile_class_posix_loop:w #1#2
19552 {
19553   \token_if_eq_meaning:NNTF \_regex_compile_raw:N #1
19554   { #2 \_regex_compile_class_posix_loop:w }
19555   { \if_false: { \fi: } \_regex_compile_class_posix_end:w #1 #2 }
19556 }
19557 \cs_new_protected:Npn \_regex_compile_class_posix_end:w #1#2#3#4
19558 {
19559   \str_if_eq:nnTF { #1 #2 #3 #4 }
19560   { \_regex_compile_special:N : \_regex_compile_special:N ] }
19561   {
19562     \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
19563     {
19564       \_regex_compile_one:n
19565       {

```



```

19566         \bool_if:NF \l__regex_internal_bool \__regex_item_reverse:n
19567         \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : }
19568     }
19569 }
19570 {
19571     \__kernel_msg_warning:nxx { kernel } { posix-unknown }
19572     { \l__regex_internal_a_tl }
19573     \__regex_compile_abort_tokens:x
19574     {
19575         [: \bool_if:NF \l__regex_internal_bool { ^ }
19576         \l__regex_internal_a_tl :]
19577     }
19578 }
19579 }
19580 {
19581     \__kernel_msg_error:nxxx { kernel } { posix-missing-close }
19582     { [: \l__regex_internal_a_tl ] { #2 #4 }
19583     \__regex_compile_abort_tokens:x { [: \l__regex_internal_a_tl ]
19584     #1 #2 #3 #4
19585     }
19586 }

```

(End definition for `__regex_compile_class_posix_test:w` and others.)

37.3.10 Groups and alternations

`__regex_compile_group_begin:N` The contents of a regex group are turned into compiled code in `\l__regex_build_tl`, which ends up with items of the form `__regex_branch:n {⟨concatenation⟩}`. This construction is done using `\tl_build_...` functions within a `TeX` group, which automatically makes sure that options (case-sensitivity and default catcode) are reset at the end of the group. The argument `#1` is `__regex_group:nnnN` or a variant thereof. A small subtlety to support `\cL(abc)` as a shorthand for `(\cLa\cLb\cLc)`: exit any pending catcode test, save the category code at the start of the group as the default catcode for that group, and make sure that the catcode is restored to the default outside the group.

```

19587 \cs_new_protected:Npn \__regex_compile_group_begin:N #1
19588 {
19589     \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
19590     \__regex_mode_quit_c:
19591     \group_begin:
19592         \tl_build_begin:N \l__regex_build_tl
19593         \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
19594         \int_incr:N \l__regex_group_level_int
19595         \tl_build_put_right:Nn \l__regex_build_tl
19596         { \__regex_branch:n { \if_false: } \fi: }
19597     }
19598 \cs_new_protected:Npn \__regex_compile_group_end:
19599 {
19600     \if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f:
19601         \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
19602         \tl_build_end:N \l__regex_build_tl
19603         \exp_args:NNNx
19604         \group_end:

```

```

19605 \tl_build_put_right:Nn \l__regex_build_tl { \l__regex_build_tl }
19606 \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
19607 \exp_after:wN \__regex_compile_quantifier:w
19608 \else:
19609 \__kernel_msg_warning:nn { kernel } { extra-rparen }
19610 \exp_after:wN \__regex_compile_raw:N \exp_after:wN )
19611 \fi:
19612 }

```

(End definition for __regex_compile_group_begin:N and __regex_compile_group_end:.)

__regex_compile(: In a class, parentheses are not special. In a catcode test inside a class, a left parenthesis gives an error, to catch [a\cL(bcd)e]. Otherwise check for a ?, denoting special groups, and run the code for the corresponding special group.

```

19613 \cs_new_protected:cpn { __regex_compile(: }
19614 {
19615 \__regex_if_in_class:TF { \__regex_compile_raw:N ( }
19616 {
19617 \if_int_compare:w \l__regex_mode_int =
19618 \c__regex_catcode_in_class_mode_int
19619 \__kernel_msg_error:nn { kernel } { c-lparen-in-class }
19620 \exp_after:wN \__regex_compile_raw:N \exp_after:wN (
19621 \else:
19622 \exp_after:wN \__regex_compile_lparen:w
19623 \fi:
19624 }
19625 }
19626 \cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
19627 {
19628 \str_if_eq:nnTF { #1 #2 } { \__regex_compile_special:N ? }
19629 {
19630 \cs_if_exist_use:cF
19631 { __regex_compile_special_group\_token_to_str:N #4 :w }
19632 {
19633 \__kernel_msg_warning:nnx { kernel } { special-group-unknown }
19634 { (? #4 }
19635 \__regex_compile_group_begin:N \__regex_group:nnnN
19636 \__regex_compile_raw:N ? #3 #4
19637 }
19638 }
19639 {
19640 \__regex_compile_group_begin:N \__regex_group:nnnN
19641 #1 #2 #3 #4
19642 }
19643 }

```

(End definition for __regex_compile(:.)

__regex_compile_|: In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```

19644 \cs_new_protected:cpn { __regex_compile_|: }
19645 {
19646 \__regex_if_in_class:TF { \__regex_compile_raw:N | }
19647 {

```

```

19648         \tl_build_put_right:Nn \l__regex_build_tl
19649         { \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
19650     }
19651 }

```

(End definition for __regex_compile_l:.)

__regex_compile_): Within a class, parentheses are not special. Outside, close a group.

```

19652 \cs_new_protected:cpn { __regex_compile_): }
19653 {
19654     \__regex_if_in_class:TF { \__regex_compile_raw:N ) }
19655     { \__regex_compile_group_end: }
19656 }

```

(End definition for __regex_compile_):.)

_regex_compile_special_group_:w Non-capturing, and resetting groups are easy to take care of during compilation; for those groups, the harder parts come when building.

```

19657 \cs_new_protected:cpn { __regex_compile_special_group_:w }
19658 { \__regex_compile_group_begin:N \__regex_group_no_capture:nnnN }
19659 \cs_new_protected:cpn { __regex_compile_special_group_|:w }
19660 { \__regex_compile_group_begin:N \__regex_group_resetting:nnnN }

```

(End definition for __regex_compile_special_group_:w and __regex_compile_special_group_|:w.)

_regex_compile_special_group_i:w The match can be made case-insensitive by setting the option with (?i); the original behaviour is restored by (?-i). This is the only supported option.

```

19661 \cs_new_protected:Npn \__regex_compile_special_group_i:w #1#2
19662 {
19663     \str_if_eq:nnTF { #1 #2 } { \__regex_compile_special:N ) }
19664     {
19665         \cs_set:Npn \__regex_item_equal:n
19666         { \__regex_item_caseless_equal:n }
19667         \cs_set:Npn \__regex_item_range:nn
19668         { \__regex_item_caseless_range:nn }
19669     }
19670     {
19671         \__kernel_msg_warning:nnx { kernel } { unknown-option } { (?i #2 }
19672         \__regex_compile_raw:N (
19673         \__regex_compile_raw:N ?
19674         \__regex_compile_raw:N i
19675         #1 #2
19676     }
19677 }
19678 \cs_new_protected:cpn { __regex_compile_special_group_-:w } #1#2#3#4
19679 {
19680     \str_if_eq:nnTF { #1 #2 #3 #4 }
19681     { \__regex_compile_raw:N i \__regex_compile_special:N ) }
19682     {
19683         \cs_set:Npn \__regex_item_equal:n
19684         { \__regex_item_caseful_equal:n }
19685         \cs_set:Npn \__regex_item_range:nn
19686         { \__regex_item_caseful_range:nn }
19687     }
19688     {

```

```

19689         \_kernel_msg_warning:nxx { kernel } { unknown-option } { (?-#2#4 }
19690         \_regex_compile_raw:N (
19691         \_regex_compile_raw:N ?
19692         \_regex_compile_raw:N -
19693         #1 #2 #3 #4
19694     }
19695 }

```

(End definition for _regex_compile_special_group_i:w and _regex_compile_special_group_~:w.)

37.3.11 Catcodes and csnames

_regex_compile_/c: The \c escape sequence can be followed by a capital letter representing a character category, by a left bracket which starts a list of categories, or by a brace group holding a regular expression for a control sequence name. Otherwise, raise an error.

```

19696 \cs_new_protected:cpn { \_regex_compile_/c: }
19697 { \_regex_chk_c_allowed:T { \_regex_compile_c_test:NN } }
19698 \cs_new_protected:Npn \_regex_compile_c_test:NN #1#2
19699 {
19700     \token_if_eq_meaning:NNTF #1 \_regex_compile_raw:N
19701     {
19702         \int_if_exist:cTF { c__regex_catcode_#2_int }
19703         {
19704             \int_set_eq:Nc \l__regex_catcodes_int
19705             { c__regex_catcode_#2_int }
19706             \l__regex_mode_int
19707             = \if_case:w \l__regex_mode_int
19708               \c__regex_catcode_mode_int
19709             \else:
19710               \c__regex_catcode_in_class_mode_int
19711             \fi:
19712             \token_if_eq_charcode:NNT C #2 { \_regex_compile_c_C:NN }
19713         }
19714     }
19715     { \cs_if_exist_use:cF { \_regex_compile_c_#2:w } }
19716     {
19717         \_kernel_msg_error:nxx { kernel } { c-missing-category } {#2}
19718         #1 #2
19719     }
19720 }

```

(End definition for _regex_compile_/c: and _regex_compile_c_test:NN.)

_regex_compile_c_C:NN If \cC is not followed by . or (...) then complain because that construction cannot match anything, except in cases like \cC[\c{...}], where it has no effect.

```

19721 \cs_new_protected:Npn \_regex_compile_c_C:NN #1#2
19722 {
19723     \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
19724     {
19725         \token_if_eq_charcode:NNTF #2 .
19726         { \use_none:n }
19727         { \token_if_eq_charcode:NNTF #2 ( } % )
19728     }
19729     { \use_n:n }

```

```

19730     { \__kernel_msg_error:nnn { kernel } { c-C-invalid } {#2} }
19731     #1 #2
19732   }

```

(End definition for __regex_compile_c:C:NN.)

__regex_compile_c[:w When encountering \c[, the task is to collect uppercase letters representing character categories. First check for ^ which negates the list of category codes.

```

\__regex_compile_c_lbrack_loop:NN
\__regex_compile_c_lbrack_add:N
\__regex_compile_c_lbrack_end:
19733 \cs_new_protected:cpn { \__regex_compile_c[:w } #1#2
19734   {
19735     \l__regex_mode_int
19736     = \if_case:w \l__regex_mode_int
19737       \c__regex_catcode_mode_int
19738       \else:
19739         \c__regex_catcode_in_class_mode_int
19740       \fi:
19741     \int_zero:N \l__regex_catcodes_int
19742     \str_if_eq:nnTF { #1 #2 } { \__regex_compile_special:N ^ }
19743     {
19744       \bool_set_false:N \l__regex_catcodes_bool
19745       \__regex_compile_c_lbrack_loop:NN
19746     }
19747     {
19748       \bool_set_true:N \l__regex_catcodes_bool
19749       \__regex_compile_c_lbrack_loop:NN
19750       #1 #2
19751     }
19752   }
19753 \cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2
19754   {
19755     \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
19756     {
19757       \int_if_exist:cTF { c__regex_catcode_#2_int }
19758       {
19759         \exp_args:Nc \__regex_compile_c_lbrack_add:N
19760         { c__regex_catcode_#2_int }
19761         \__regex_compile_c_lbrack_loop:NN
19762       }
19763     }
19764     {
19765       \token_if_eq_charcode:NNTF #2 ]
19766       { \__regex_compile_c_lbrack_end: }
19767     }
19768     {
19769       \__kernel_msg_error:nnx { kernel } { c-missing-rbrack } {#2}
19770       \__regex_compile_c_lbrack_end:
19771       #1 #2
19772     }
19773   }
19774 \cs_new_protected:Npn \__regex_compile_c_lbrack_add:N #1
19775   {
19776     \if_int_odd:w \int_eval:n { \l__regex_catcodes_int / #1 } \exp_stop_f:
19777     \else:
19778       \int_add:Nn \l__regex_catcodes_int {#1}

```

```

19779     \fi:
19780   }
19781 \cs_new_protected:Npn \__regex_compile_c_lbrack_end:
19782   {
19783     \if_meaning:w \c_false_bool \l__regex_catcodes_bool
19784       \int_set:Nn \l__regex_catcodes_int
19785         { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
19786     \fi:
19787   }

```

(End definition for __regex_compile_c[:w and others.)

__regex_compile_c_{: The case of a left brace is easy, based on what we have done so far: in a group, compile the regular expression, after changing the mode to forbid nesting \c. Additionally, disable submatch tracking since groups don't escape the scope of \c{...}.

```

19788 \cs_new_protected:cpn { __regex_compile_c_ \c_left_brace_str :w }
19789   {
19790     \__regex_compile:w
19791     \__regex_disable_submatches:
19792     \l__regex_mode_int
19793     = \if_case:w \l__regex_mode_int
19794       \c__regex_cs_mode_int
19795     \else:
19796       \c__regex_cs_in_class_mode_int
19797     \fi:
19798   }

```

(End definition for __regex_compile_c{:.)

__regex_compile_}: Non-escaped right braces are only special if they appear when compiling the regular expression for a csname, but not within a class: \c{[{}]} matches the control sequences \{ and \}. So, end compiling the inner regex (this closes any dangling class or group). Then insert the corresponding test in the outer regex. As an optimization, if the control sequence test simply consists of several explicit possibilities (branches) then use __regex_item_exact_cs:n with an argument consisting of all possibilities separated by \scan_stop:.

```

19799 \flag_new:n { __regex_cs }
19800 \cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
19801   {
19802     \__regex_if_in_cs:TF
19803       { \__regex_compile_end_cs: }
19804       { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
19805   }
19806 \cs_new_protected:Npn \__regex_compile_end_cs:
19807   {
19808     \__regex_compile_end:
19809     \flag_clear:n { __regex_cs }
19810     \tl_set:Nx \l__regex_internal_a_tl
19811       {
19812         \exp_after:wN \__regex_compile_cs_aux:Nn \l__regex_internal_regex
19813         \q_nil \q_nil \q_recursion_stop
19814       }
19815     \exp_args:Nx \__regex_compile_one:n
19816     {

```

```

19817     \flag_if_raised:nTF { __regex_cs }
19818     { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
19819     {
19820         \__regex_item_exact_cs:n
19821         { \tl_tail:N \l__regex_internal_a_tl }
19822     }
19823 }
19824 }
19825 \cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
19826 {
19827     \cs_if_eq:NNTF #1 \__regex_branch:n
19828     {
19829         \scan_stop:
19830         \__regex_compile_cs_aux:NNnnN #2
19831         \q_nil \q_nil \q_nil \q_nil \q_nil \q_recursion_stop
19832         \__regex_compile_cs_aux:Nn
19833     }
19834     {
19835         \quark_if_nil:NF #1 { \flag_raise_if_clear:n { __regex_cs } }
19836         \use_none_delimit_by_q_recursion_stop:w
19837     }
19838 }
19839 \cs_new:Npn \__regex_compile_cs_aux:NNnnnN #1#2#3#4#5#6
19840 {
19841     \bool_lazy_all:nTF
19842     {
19843         { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
19844         {#2}
19845         { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
19846         { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
19847         { \int_compare_p:nNn {#5} = { 0 } }
19848     }
19849     {
19850         \prg_replicate:nn {#4}
19851         { \char_generate:nn { \use_ii:nn #3 } {12} }
19852         \__regex_compile_cs_aux:NNnnnN
19853     }
19854     {
19855         \quark_if_nil:NF #1
19856         {
19857             \flag_raise_if_clear:n { __regex_cs }
19858             \use_i_delimit_by_q_recursion_stop:nw
19859         }
19860         \use_none_delimit_by_q_recursion_stop:w
19861     }
19862 }

```

(End definition for __regex_compile_}: and others.)

37.3.12 Raw token lists with \u

__regex_compile_/u: The \u escape is invalid in classes and directly following a catcode test. Otherwise, it must be followed by a left brace. We then collect the characters for the argument of \u within an x-expanding assignment. In principle we could just wait to encounter a

right brace, but this is unsafe: if the right brace was missing, then we would reach the end-markers of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace, any escaped character, or the end-marker.

```

19863 \cs_new_protected:cpn { __regex_compile_/u: } #1#2
19864 {
19865   \__regex_if_in_class_or_catcode:TF
19866   { \__regex_compile_raw_error:N u #1 #2 }
19867   {
19868     \str_if_eq_x:nnTF {#1#2}
19869     { \__regex_compile_special:N \c_left_brace_str }
19870     {
19871       \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
19872       \__regex_compile_u_loop:NN
19873     }
19874     {
19875       \__kernel_msg_error:nn { kernel } { u-missing-lbrace }
19876       \__regex_compile_raw:N u #1 #2
19877     }
19878   }
19879 }
19880 \cs_new:Npn \__regex_compile_u_loop:NN #1#2
19881 {
19882   \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
19883   { #2 \__regex_compile_u_loop:NN }
19884   {
19885     \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
19886     {
19887       \exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2
19888       { \if_false: { \fi: } \__regex_compile_u_end: }
19889       { #2 \__regex_compile_u_loop:NN }
19890     }
19891     {
19892       \if_false: { \fi: }
19893       \__kernel_msg_error:nnx { kernel } { u-missing-rbrace } {#2}
19894       \__regex_compile_u_end:
19895       #1 #2
19896     }
19897   }
19898 }

```

(End definition for __regex_compile_/u: and __regex_compile_u_loop:NN.)

__regex_compile_u_end: Once we have extracted the variable's name, we store the contents of that variable in \l__regex_internal_a_tl. The behaviour of \u then depends on whether we are within a \c{...} escape (in this case, the variable is turned to a string), or not.

```

19899 \cs_new_protected:Npn \__regex_compile_u_end:
19900 {
19901   \tl_set:Nv \l__regex_internal_a_tl { \l__regex_internal_a_tl }
19902   \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
19903   \__regex_compile_u_not_cs:
19904   \else:
19905     \__regex_compile_u_in_cs:
19906   \fi:

```



```
19907     }
```

(End definition for `_regex_compile_u_end:.`)

`_regex_compile_u_in_cs:` When `\u` appears within a control sequence, we convert the variable to a string with escaped spaces. Then for each character insert a class matching exactly that character, once.

```
19908 \cs_new_protected:Npn \_regex_compile_u_in_cs:
19909 {
19910     \tl_gset:Nx \g__regex_internal_tl
19911     {
19912         \exp_args:No \_kernel_str_to_other_fast:n
19913         { \l__regex_internal_a_tl }
19914     }
19915     \tl_build_put_right:Nx \l__regex_build_tl
19916     {
19917         \tl_map_function:NN \g__regex_internal_tl
19918         \_regex_compile_u_in_cs_aux:n
19919     }
19920 }
19921 \cs_new:Npn \_regex_compile_u_in_cs_aux:n #1
19922 {
19923     \_regex_class:NnnnN \c_true_bool
19924     { \_regex_item_caseful_equal:n { \int_value:w '#1 } }
19925     { 1 } { 0 } \c_false_bool
19926 }
```

(End definition for `_regex_compile_u_in_cs:.`)

`_regex_compile_u_not_cs:` In mode 0, the `\u` escape adds one state to the NFA for each token in `\l__regex_internal_a_tl`. If a given *<token>* is a control sequence, then insert a string comparison test, otherwise, `_regex_item_exact:nn` which compares catcode and character code.

```
19927 \cs_new_protected:Npn \_regex_compile_u_not_cs:
19928 {
19929     \tl_analysis_map_inline:Nn \l__regex_internal_a_tl
19930     {
19931         \tl_build_put_right:Nx \l__regex_build_tl
19932         {
19933             \_regex_class:NnnnN \c_true_bool
19934             {
19935                 \if_int_compare:w "##3 = 0 \exp_stop_f:
19936                 \_regex_item_exact_cs:n
19937                 { \exp_after:wN \cs_to_str:N ##1 }
19938                 \else:
19939                 \_regex_item_exact:nn { \int_value:w "##3 } { ##2 }
19940                 \fi:
19941             }
19942             { 1 } { 0 } \c_false_bool
19943         }
19944     }
19945 }
```

(End definition for `_regex_compile_u_not_cs:.`)

37.3.13 Other

`__regex_compile_/K:` The `\K` control sequence is currently the only “command”, which performs some action, rather than matching something. It is allowed in the same contexts as `\b`. At the compilation stage, we leave it as a single control sequence, defined later.

```

19946 \cs_new_protected:cpn { __regex_compile_/K: }
19947 {
19948   \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
19949     { \tl_build_put_right:Nn \l__regex_build_tl { \__regex_command_K: } }
19950     { \__regex_compile_raw_error:N K }
19951 }

```

(End definition for `__regex_compile_/K:.`)

37.3.14 Showing regexes

`__regex_show:N` Within a group and within `\tl_build_begin:N ... \tl_build_end:N` we redefine all the function that can appear in a compiled regex, then run the regex. The result stored in `\l__regex_internal_a_tl` is then meant to be shown.

```

19952 \cs_new_protected:Npn \__regex_show:N #1
19953 {
19954   \group_begin:
19955     \tl_build_begin:N \l__regex_build_tl
19956     \cs_set_protected:Npn \__regex_branch:n
19957     {
19958       \seq_pop_right:NN \l__regex_show_prefix_seq
19959       \l__regex_internal_a_tl
19960       \__regex_show_one:n { +-branch }
19961       \seq_put_right:No \l__regex_show_prefix_seq
19962       \l__regex_internal_a_tl
19963       \use:n
19964     }
19965     \cs_set_protected:Npn \__regex_group:nnnN
19966     { \__regex_show_group_aux:nnnnN { } }
19967     \cs_set_protected:Npn \__regex_group_no_capture:nnnN
19968     { \__regex_show_group_aux:nnnnN { ~(no~capture) } }
19969     \cs_set_protected:Npn \__regex_group_resetting:nnnN
19970     { \__regex_show_group_aux:nnnnN { ~(resetting) } }
19971     \cs_set_eq:NN \__regex_class:NnnnN \__regex_show_class:NnnnN
19972     \cs_set_protected:Npn \__regex_command_K:
19973     { \__regex_show_one:n { reset~match~start~(\iow_char:N\K) } }
19974     \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
19975     {
19976       \__regex_show_one:n
19977       { \bool_if:NF ##1 { negative~ } assertion:~##2 }
19978     }
19979     \cs_set:Npn \__regex_b_test: { word~boundary }
19980     \cs_set_eq:NN \__regex_anchor:N \__regex_show_anchor_to_str:N
19981     \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
19982     { \__regex_show_one:n { char~code~\int_eval:n{##1} } }
19983     \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
19984     {
19985       \__regex_show_one:n
19986       { range~[\int_eval:n{##1}, \int_eval:n{##2}] }

```

```

19987     }
19988     \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
19989     { \__regex_show_one:n { char-code~\int_eval:n{##1}~(caseless) } }
19990     \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
19991     {
19992         \__regex_show_one:n
19993         { Range~[\int_eval:n{##1}, \int_eval:n{##2}]~(caseless) }
19994     }
19995     \cs_set_protected:Npn \__regex_item_catcode:nT
19996     { \__regex_show_item_catcode:NnT \c_true_bool }
19997     \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
19998     { \__regex_show_item_catcode:NnT \c_false_bool }
19999     \cs_set_protected:Npn \__regex_item_reverse:n
20000     { \__regex_show_scope:nn { Reversed~match } }
20001     \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
20002     { \__regex_show_one:n { char~##2,~catcode~##1 } }
20003     \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
20004     \cs_set_protected:Npn \__regex_item_cs:n
20005     { \__regex_show_scope:nn { control~sequence } }
20006     \cs_set:cpn { __regex_prop_.: } { \__regex_show_one:n { any~token } }
20007     \seq_clear:N \l__regex_show_prefix_seq
20008     \__regex_show_push:n { ~ }
20009     \cs_if_exist_use:N #1
20010     \tl_build_end:N \l__regex_build_tl
20011     \exp_args:NNNo
20012     \group_end:
20013     \tl_set:Nn \l__regex_internal_a_tl { \l__regex_build_tl }
20014 }

```

(End definition for __regex_show:N.)

__regex_show_one:n Every part of the final message go through this function, which adds one line to the output, with the appropriate prefix.

```

20015 \cs_new_protected:Npn \__regex_show_one:n #1
20016 {
20017     \int_incr:N \l__regex_show_lines_int
20018     \tl_build_put_right:Nx \l__regex_build_tl
20019     {
20020         \exp_not:N \iow_newline:
20021         \seq_map_function:NN \l__regex_show_prefix_seq \use:n
20022         #1
20023     }
20024 }

```

(End definition for __regex_show_one:n.)

__regex_show_push:n Enter and exit levels of nesting. The scope function prints its first argument as an “introduction”, then performs its second argument in a deeper level of nesting.

```

\__regex_show_pop:
\__regex_show_scope:nn
20025 \cs_new_protected:Npn \__regex_show_push:n #1
20026 { \seq_put_right:Nx \l__regex_show_prefix_seq { #1 ~ } }
20027 \cs_new_protected:Npn \__regex_show_pop:
20028 { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
20029 \cs_new_protected:Npn \__regex_show_scope:nn #1#2
20030 {

```

```

20031     \__regex_show_one:n {#1}
20032     \__regex_show_push:n { ~ }
20033     #2
20034     \__regex_show_pop:
20035 }

```

(End definition for __regex_show_push:n, __regex_show_pop:, and __regex_show_scope:nn.)

__regex_show_group_aux:nnnnN

We display all groups in the same way, simply adding a message, (no capture) or (resetting), to special groups. The odd \use_ii:nn avoids printing a spurious +-branch for the first branch.

```

20036 \cs_new_protected:Npn \__regex_show_group_aux:nnnnN #1#2#3#4#5
20037 {
20038     \__regex_show_one:n { ,-group-begin #1 }
20039     \__regex_show_push:n { | }
20040     \use_ii:nn #2
20041     \__regex_show_pop:
20042     \__regex_show_one:n
20043     { '-group-end \__regex_msg_repeated:nnN {#3} {#4} #5 }
20044 }

```

(End definition for __regex_show_group_aux:nnnnN.)

__regex_show_class:NnnnN

I'm entirely unhappy about this function: I couldn't find a way to test if a class is a single test. Instead, collect the representation of the tests in the class. If that had more than one line, write Match or Don't match on its own line, with the repeating information if any. Then the various tests on lines of their own, and finally a line. Otherwise, we need to evaluate the representation of the tests again (since the prefix is incorrect). That's clunky, but not too expensive, since it's only one test.

```

20045 \cs_set:Npn \__regex_show_class:NnnnN #1#2#3#4#5
20046 {
20047     \group_begin:
20048     \tl_build_begin:N \l__regex_build_tl
20049     \int_zero:N \l__regex_show_lines_int
20050     \__regex_show_push:n {~}
20051     #2
20052     \int_compare:nTF { \l__regex_show_lines_int = 0 }
20053     {
20054         \group_end:
20055         \__regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
20056     }
20057     {
20058         \bool_if:nTF
20059         { #1 && \int_compare_p:n { \l__regex_show_lines_int = 1 } }
20060         {
20061             \group_end:
20062             #2
20063             \tl_build_put_right:Nn \l__regex_build_tl
20064             { \__regex_msg_repeated:nnN {#3} {#4} #5 }
20065         }
20066         {
20067             \tl_build_end:N \l__regex_build_tl
20068             \exp_args:NNNo
20069             \group_end:

```

```

20070         \tl_set:Nn \l__regex_internal_a_tl \l__regex_build_tl
20071         \__regex_show_one:n
20072         {
20073             \bool_if:NTF #1 { Match } { Don't-match }
20074             \__regex_msg_repeated:nnN {#3} {#4} #5
20075         }
20076         \tl_build_put_right:Nx \l__regex_build_tl
20077         { \exp_not:o \l__regex_internal_a_tl }
20078     }
20079 }
20080 }

```

(End definition for __regex_show_class:NnnnN.)

__regex_show_anchor_to_str:N The argument is an integer telling us where the anchor is. We convert that to the relevant info.

```

20081 \cs_new:Npn \__regex_show_anchor_to_str:N #1
20082 {
20083     anchor~at~
20084     \str_case:nnF { #1 }
20085     {
20086         { \l__regex_min_pos_int } { start~(\iow_char:N\A) }
20087         { \l__regex_start_pos_int } { start~of~match~(\iow_char:N\G) }
20088         { \l__regex_max_pos_int } { end~(\iow_char:N\Z) }
20089     }
20090     { <error:~'1'~not~recognized> }
20091 }

```

(End definition for __regex_show_anchor_to_str:N.)

__regex_show_item_catcode:NnT Produce a sequence of categories which the catcode bitmap #2 contains, and show it, indenting the tests on which this catcode constraint applies.

```

20092 \cs_new_protected:Npn \__regex_show_item_catcode:NnT #1#2
20093 {
20094     \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
20095     \seq_set_filter:NNn \l__regex_internal_seq \l__regex_internal_seq
20096     { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
20097     \__regex_show_scope:nn
20098     {
20099         categories~
20100         \seq_map_function:NN \l__regex_internal_seq \use:n
20101         , ~
20102         \bool_if:NF #1 { negative~ } class
20103     }
20104 }

```

(End definition for __regex_show_item_catcode:NnT.)

__regex_show_item_exact_cs:n

```

20105 \cs_new_protected:Npn \__regex_show_item_exact_cs:n #1
20106 {
20107     \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
20108     \seq_set_map:NNn \l__regex_internal_seq
20109     \l__regex_internal_seq { \iow_char:N\##1 }
20110     \__regex_show_one:n

```

```

20111         { control~sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
20112     }

```

(End definition for `__regex_show_item_exact_cs:n`.)

37.4 Building

37.4.1 Variables used while building

`\l__regex_min_state_int` The last state that was allocated is `\l__regex_max_state_int - 1`, so that `\l__regex_max_state_int` always points to a free state. The `min_state` variable is 1, but is included to avoid hard-coding this value everywhere.

```

20113 \int_new:N \l__regex_min_state_int
20114 \int_set:Nn \l__regex_min_state_int { 1 }
20115 \int_new:N \l__regex_max_state_int

```

(End definition for `\l__regex_min_state_int` and `\l__regex_max_state_int`.)

`\l__regex_left_state_int` Alternatives are implemented by branching from a `left` state into the various choices, then merging those into a `right` state. We store information about those states in two sequences. Those states are also used to implement group quantifiers. Most often, the `\l__regex_right_state_int` left and right pointers only differ by 1.

```

20116 \int_new:N \l__regex_left_state_int
20117 \int_new:N \l__regex_right_state_int
20118 \seq_new:N \l__regex_left_state_seq
20119 \seq_new:N \l__regex_right_state_seq

```

(End definition for `\l__regex_left_state_int` and others.)

`\l__regex_capturing_group_int` `\l__regex_capturing_group_int` is the next ID number to be assigned to a capturing group. This starts at 0 for the group enclosing the full regular expression, and groups are counted in the order of their left parenthesis, except when encountering `resetting` groups.

```

20120 \int_new:N \l__regex_capturing_group_int

```

(End definition for `\l__regex_capturing_group_int`.)

37.4.2 Framework

This phase is about going from a compiled regex to an NFA. Each state of the NFA is stored in a `\toks`. The operations which can appear in the `\toks` are

- `__regex_action_start_wildcard`: inserted at the start of the regular expression to make it unanchored.
- `__regex_action_success`: marks the exit state of the NFA.
- `__regex_action_cost:n {⟨shift⟩}` is a transition from the current `⟨state⟩` to `⟨state⟩ + ⟨shift⟩`, which consumes the current character: the target state is saved and will be considered again when matching at the next position.
- `__regex_action_free:n {⟨shift⟩}`, and `__regex_action_free_group:n {⟨shift⟩}` are free transitions, which immediately perform the actions for the state `⟨state⟩ + ⟨shift⟩` of the NFA. They differ in how they detect and avoid infinite loops. For now, we just need to know that the `group` variant must be used for transitions back to the start of a group.

- `__regex_action_submatch:n {⟨key⟩}` where the `⟨key⟩` is a group number followed by `<` or `>` for the beginning or end of group. This causes the current position in the query to be stored as the `⟨key⟩` submatch boundary.

We strive to preserve the following properties while building.

- The current capturing group is `capturing_group - 1`, and if a group opened now it would be labelled `capturing_group`.
- The last allocated state is `max_state - 1`, so `max_state` is a free state.
- The `left_state` points to a state to the left of the current group or of the last class.
- The `right_state` points to a newly created, empty state, with some transitions leading to it.
- The `left/right` sequences hold a list of the corresponding end-points of nested groups.

`__regex_build:n` The `n`-type function first compiles its argument. Reset some variables. Allocate two states, and put a wildcard in state 0 (transitions to state 1 and 0 state). Then build the regex within a (capturing) group numbered 0 (current value of `capturing_group`). Finally, if the match reaches the last state, it is successful.

```

20121 \cs_new_protected:Npn \__regex_build:n #1
20122 {
20123   \__regex_compile:n {#1}
20124   \__regex_build:N \l__regex_internal_regex
20125 }
20126 \__kernel_patch:nnNNpn
20127 { \__regex_trace_push:nnN { regex } { 1 } \__regex_build:N }
20128 {
20129   \__regex_trace_states:n { 2 }
20130   \__regex_trace_pop:nnN { regex } { 1 } \__regex_build:N
20131 }
20132 \cs_new_protected:Npn \__regex_build:N #1
20133 {
20134   \__regex_standard_escapechar:
20135   \int_zero:N \l__regex_capturing_group_int
20136   \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
20137   \__regex_build_new_state:
20138   \__regex_build_new_state:
20139   \__regex_toks_put_right:Nn \l__regex_left_state_int
20140     { \__regex_action_start_wildcard: }
20141   \__regex_group:nnnN {#1} { 1 } { 0 } \c_false_bool
20142   \__regex_toks_put_right:Nn \l__regex_right_state_int
20143     { \__regex_action_success: }
20144 }

```

(End definition for `__regex_build:n` and `__regex_build:N`.)

`__regex_build_for_cs:n` When using a regex to match a `cs`, we don't insert a wildcard, we anchor at the end, and since we ignore submatches, there is no need to surround the expression with a group. However, for branches to work properly at the outer level, we need to put the appropriate `left` and `right` states in their sequence.

```

20145 \__kernel_patch:nnNNpn
20146 { \__regex_trace_push:nnN { regex } { 1 } \__regex_build_for_cs:n }
20147 {
20148   \__regex_trace_states:n { 2 }
20149   \__regex_trace_pop:nnN { regex } { 1 } \__regex_build_for_cs:n
20150 }
20151 \cs_new_protected:Npn \__regex_build_for_cs:n #1
20152 {
20153   \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
20154   \__regex_build_new_state:
20155   \__regex_build_new_state:
20156   \__regex_push_lr_states:
20157   #1
20158   \__regex_pop_lr_states:
20159   \__regex_toks_put_right:Nn \l__regex_right_state_int
20160   {
20161     \if_int_compare:w \l__regex_curr_pos_int = \l__regex_max_pos_int
20162       \exp_after:wN \__regex_action_success:
20163     \fi:
20164   }
20165 }

```

(End definition for __regex_build_for_cs:n.)

37.4.3 Helpers for building an nfa

__regex_push_lr_states: When building the regular expression, we keep track of pointers to the left-end and
 __regex_pop_lr_states: right-end of each group without help from T_EX's grouping.

```

20166 \cs_new_protected:Npn \__regex_push_lr_states:
20167 {
20168   \seq_push:No \l__regex_left_state_seq
20169   { \int_use:N \l__regex_left_state_int }
20170   \seq_push:No \l__regex_right_state_seq
20171   { \int_use:N \l__regex_right_state_int }
20172 }
20173 \cs_new_protected:Npn \__regex_pop_lr_states:
20174 {
20175   \seq_pop:NN \l__regex_left_state_seq \l__regex_internal_a_tl
20176   \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
20177   \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
20178   \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
20179 }

```

(End definition for __regex_push_lr_states: and __regex_pop_lr_states:.)

__regex_build_transition_left:NNN Add a transition from #2 to #3 using the function #1. The left function is used for
 __regex_build_transition_right:nNn higher priority transitions, and the right function for lower priority transitions (which
 should be performed later). The signatures differ to reflect the differing usage later on.
 Both functions could be optimized.

```

20180 \cs_new_protected:Npn \__regex_build_transition_left:NNN #1#2#3
20181 { \__regex_toks_put_left:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }
20182 \cs_new_protected:Npn \__regex_build_transition_right:nNn #1#2#3
20183 { \__regex_toks_put_right:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }

```

(End definition for __regex_build_transition_left:NNN and __regex_build_transition_right:nNn.)

`__regex_build_new_state:` Add a new empty state to the NFA. Then update the `left`, `right`, and `max` states, so that the `right` state is the new empty state, and the `left` state points to the previously “current” state.

```

20184 \__kernel_patch:nnNNpn
20185 {
20186   \__regex_trace:nnx { regex } { 2 }
20187   {
20188     regex~new~state~
20189     L=\int_use:N \l__regex_left_state_int ~ -> ~
20190     R=\int_use:N \l__regex_right_state_int ~ -> ~
20191     M=\int_use:N \l__regex_max_state_int ~ -> ~
20192     \int_eval:n { \l__regex_max_state_int + 1 }
20193   }
20194 }
20195 { }
20196 \cs_new_protected:Npn \__regex_build_new_state:
20197 {
20198   \__regex_toks_clear:N \l__regex_max_state_int
20199   \int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
20200   \int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
20201   \int_incr:N \l__regex_max_state_int
20202 }

```

(End definition for `__regex_build_new_state:.`)

`__regex_build_transitions_lazyiness:NNNNN` This function creates a new state, and puts two transitions starting from the old current state. The order of the transitions is controlled by `#1`, true for lazy quantifiers, and false for greedy quantifiers.

```

20203 \cs_new_protected:Npn \__regex_build_transitions_lazyiness:NNNNN #1#2#3#4#5
20204 {
20205   \__regex_build_new_state:
20206   \__regex_toks_put_right:Nx \l__regex_left_state_int
20207   {
20208     \if_meaning:w \c_true_bool #1
20209       #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
20210       #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
20211     \else:
20212       #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
20213       #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
20214     \fi:
20215   }
20216 }

```

(End definition for `__regex_build_transitions_lazyiness:NNNNN.`)

37.4.4 Building classes

`__regex_class:NnnnN` The arguments are: $\langle\text{boolean}\rangle$ $\{\langle\text{tests}\rangle\}$ $\{\langle\text{min}\rangle\}$ $\{\langle\text{more}\rangle\}$ $\langle\text{lazyiness}\rangle$. First store the tests with a trailing `__regex_action_cost:n`, in the true branch of `__regex_break_point:TF` for positive classes, or the false branch for negative classes. The integer $\langle\text{more}\rangle$ is 0 for fixed repetitions, -1 for unbounded repetitions, and $\langle\text{max}\rangle - \langle\text{min}\rangle$ for a range of repetitions.

```

20217 \cs_new_protected:Npn \__regex_class:NnnnN #1#2#3#4#5

```

```

20218 {
20219   \cs_set:Npx \__regex_tests_action_cost:n ##1
20220   {
20221     \exp_not:n { \exp_not:n {#2} }
20222     \bool_if:NTF #1
20223     { \__regex_break_point:TF { \__regex_action_cost:n {##1} } { } }
20224     { \__regex_break_point:TF { } { \__regex_action_cost:n {##1} } }
20225   }
20226   \if_case:w - #4 \exp_stop_f:
20227     \__regex_class_repeat:n {#3}
20228   \or: \__regex_class_repeat:nN {#3} #5
20229   \else: \__regex_class_repeat:nnN {#3} {#4} #5
20230   \fi:
20231 }
20232 \cs_new:Npn \__regex_tests_action_cost:n { \__regex_action_cost:n }

```

(End definition for __regex_class:NnnnN and __regex_tests_action_cost:n.)

__regex_class_repeat:n This is used for a fixed number of repetitions. Build one state for each repetition, with a transition controlled by the tests that we have collected. That works just fine for #1 = 0 repetitions: nothing is built.

```

20233 \cs_new_protected:Npn \__regex_class_repeat:n #1
20234 {
20235   \prg_replicate:nn {#1}
20236   {
20237     \__regex_build_new_state:
20238     \__regex_build_transition_right:nNn \__regex_tests_action_cost:n
20239     \l__regex_left_state_int \l__regex_right_state_int
20240   }
20241 }

```

(End definition for __regex_class_repeat:n.)

__regex_class_repeat:nN This implements unbounded repetitions of a single class (e.g. the * and + quantifiers). If the minimum number #1 of repetitions is 0, then build a transition from the current state to itself governed by the tests, and a free transition to a new state (hence skipping the tests). Otherwise, call __regex_class_repeat:n for the code to match #1 repetitions, and add free transitions from the last state to the previous one, and to a new one. In both cases, the order of transitions is controlled by the laziness boolean #2.

```

20242 \cs_new_protected:Npn \__regex_class_repeat:nN #1#2
20243 {
20244   \if_int_compare:w #1 = 0 \exp_stop_f:
20245     \__regex_build_transitions_lazyness:NNNNN #2
20246     \__regex_action_free:n \l__regex_right_state_int
20247     \__regex_tests_action_cost:n \l__regex_left_state_int
20248   \else:
20249     \__regex_class_repeat:n {#1}
20250     \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
20251     \__regex_build_transitions_lazyness:NNNNN #2
20252     \__regex_action_free:n \l__regex_right_state_int
20253     \__regex_action_free:n \l__regex_internal_a_int
20254   \fi:
20255 }

```

(End definition for `_regex_class_repeat:nnN`.)

`_regex_class_repeat:nnN` We want to build the code to match from #1 to #1 + #2 repetitions. Match #1 repetitions (can be 0). Compute the final state of the next construction as a. Build #2 > 0 states, each with a transition to the next state governed by the tests, and a transition to the final state a. The computation of a is safe because states are allocated in order, starting from `max_state`.

```

20256 \cs_new_protected:Npn \_regex_class_repeat:nnN #1#2#3
20257 {
20258   \_regex_class_repeat:n {#1}
20259   \int_set:Nn \l__regex_internal_a_int
20260     { \l__regex_max_state_int + #2 - 1 }
20261   \prg_replicate:nn { #2 }
20262   {
20263     \_regex_build_transitions_lazyness:NNNNN #3
20264     \_regex_action_free:n      \l__regex_internal_a_int
20265     \_regex_tests_action_cost:n \l__regex_right_state_int
20266   }
20267 }
```

(End definition for `_regex_class_repeat:nnN`.)

37.4.5 Building groups

`_regex_group_aux:nnnnN` Arguments: $\langle label \rangle \{ \langle contents \rangle \} \{ \langle min \rangle \} \{ \langle more \rangle \} \langle lazyness \rangle$. If $\langle min \rangle$ is 0, we need to add a state before building the group, so that the thread which skips the group does not also set the start-point of the submatch. After adding one more state, the `left_state` is the left end of the group, from which all branches stem, and the `right_state` is the right end of the group, and all branches end their course in that state. We store those two integers to be queried for each branch, we build the NFA states for the contents #2 of the group, and we forget about the two integers. Once this is done, perform the repetition: either exactly #3 times, or #3 or more times, or between #3 and #3 + #4 times, with lazyness #5. The $\langle label \rangle$ #1 is used for submatch tracking. Each of the three auxiliaries expects `left_state` and `right_state` to be set properly.

```

20268 \_kernel_patch:nnNNpn
20269 { \_regex_trace_push:nnN { regex } { 1 } \_regex_group_aux:nnnnN }
20270 { \_regex_trace_pop:nnN { regex } { 1 } \_regex_group_aux:nnnnN }
20271 \cs_new_protected:Npn \_regex_group_aux:nnnnN #1#2#3#4#5
20272 {
20273   \if_int_compare:w #3 = 0 \exp_stop_f:
20274   \_regex_build_new_state:
20275   (assert)\assert_int:n { \l__regex_max_state_int = \l__regex_right_state_int + 1 }
20276   \_regex_build_transition_right:nNn \_regex_action_free_group:n
20277     \l__regex_left_state_int \l__regex_right_state_int
20278   \fi:
20279   \_regex_build_new_state:
20280   \_regex_push_lr_states:
20281   #2
20282   \_regex_pop_lr_states:
20283   \if_case:w - #4 \exp_stop_f:
20284     \_regex_group_repeat:nn {#1} {#3}
20285   \or: \_regex_group_repeat:nnN {#1} {#3} #5
20286   \else: \_regex_group_repeat:nnnn {#1} {#3} {#4} #5
```

```

20287     \fi:
20288 }

```

(End definition for `_regex_group_aux:nnnnN`.)

`_regex_group:nnnN` Hand to `_regex_group_aux:nnnnN` the label of that group (expanded), and the group itself, with some extra commands to perform.

```

20289 \cs_new_protected:Npn \_regex_group:nnnN #1
20290 {
20291     \exp_args:No \_regex_group_aux:nnnnN
20292     { \int_use:N \l__regex_capturing_group_int }
20293     {
20294         \int_incr:N \l__regex_capturing_group_int
20295         #1
20296     }
20297 }
20298 \cs_new_protected:Npn \_regex_group_no_capture:nnnN
20299 { \_regex_group_aux:nnnnN { -1 } }

```

(End definition for `_regex_group:nnnN` and `_regex_group_no_capture:nnnN`.)

`_regex_group_resetting:nnnN` Again, hand the label `-1` to `_regex_group_aux:nnnnN`, but this time we work a little bit harder to keep track of the maximum group label at the end of any branch, and to reset the group number at each branch. This relies on the fact that a compiled regex always is a sequence of items of the form `_regex_branch:n {<branch>}`.

```

20300 \cs_new_protected:Npn \_regex_group_resetting:nnnN #1
20301 {
20302     \_regex_group_aux:nnnnN { -1 }
20303     {
20304         \exp_args:Noo \_regex_group_resetting_loop:nnNn
20305         { \int_use:N \l__regex_capturing_group_int }
20306         { \int_use:N \l__regex_capturing_group_int }
20307         #1
20308         { ?? \prg_break:n } { }
20309         \prg_break_point:
20310     }
20311 }
20312 \cs_new_protected:Npn \_regex_group_resetting_loop:nnNn #1#2#3#4
20313 {
20314     \use_none:nn #3 { \int_set:Nn \l__regex_capturing_group_int {#1} }
20315     \int_set:Nn \l__regex_capturing_group_int {#2}
20316     #3 {#4}
20317     \exp_args:Nf \_regex_group_resetting_loop:nnNn
20318     { \int_max:nn {#1} { \l__regex_capturing_group_int } }
20319     {#2}
20320 }

```

(End definition for `_regex_group_resetting:nnnN` and `_regex_group_resetting_loop:nnNn`.)

`_regex_branch:n` Add a free transition from the left state of the current group to a brand new state, starting point of this branch. Once the branch is built, add a transition from its last state to the right state of the group. The left and right states of the group are extracted from the relevant sequences.

```

20321 \__kernel_patch:nnNnNpn

```

```

20322 { \_regex_trace_push:nnN { regex } { 1 } \_regex_branch:n }
20323 { \_regex_trace_pop:nnN { regex } { 1 } \_regex_branch:n }
20324 \cs_new_protected:Npn \_regex_branch:n #1
20325 {
20326   \_regex_build_new_state:
20327   \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
20328   \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
20329   \_regex_build_transition_right:nNn \_regex_action_free:n
20330     \l__regex_left_state_int \l__regex_right_state_int
20331   #1
20332   \seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_tl
20333   \_regex_build_transition_right:nNn \_regex_action_free:n
20334     \l__regex_right_state_int \l__regex_internal_a_tl
20335 }

```

(End definition for _regex_branch:n.)

_regex_group_repeat:nn This function is called to repeat a group a fixed number of times #2; if this is 0 we remove the group altogether (but don't reset the capturing_group label). Otherwise, the auxiliary _regex_group_repeat_aux:n copies #2 times the \toks for the group, and leaves internal_a pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary).

```

20336 \cs_new_protected:Npn \_regex_group_repeat:nn #1#2
20337 {
20338   \if_int_compare:w #2 = 0 \exp_stop_f:
20339     \int_set:Nn \l__regex_max_state_int
20340       { \l__regex_left_state_int - 1 }
20341     \_regex_build_new_state:
20342   \else:
20343     \_regex_group_repeat_aux:n {#2}
20344     \_regex_group_submatches:nNN {#1}
20345     \l__regex_internal_a_int \l__regex_right_state_int
20346     \_regex_build_new_state:
20347   \fi:
20348 }

```

(End definition for _regex_group_repeat:nn.)

_regex_group_submatches:nNN This inserts in states #2 and #3 the code for tracking submatches of the group #1, unless inhibited by a label of -1.

```

20349 \cs_new_protected:Npn \_regex_group_submatches:nNN #1#2#3
20350 {
20351   \if_int_compare:w #1 > - 1 \exp_stop_f:
20352     \_regex_toks_put_left:Nx #2 { \_regex_action_submatch:n { #1 < } }
20353     \_regex_toks_put_left:Nx #3 { \_regex_action_submatch:n { #1 > } }
20354   \fi:
20355 }

```

(End definition for _regex_group_submatches:nNN.)

_regex_group_repeat_aux:n Here we repeat \toks ranging from left_state to max_state, #1 > 0 times. First add a transition so that the copies “chain” properly. Compute the shift c between the original copy and the last copy we want. Shift the right_state and max_state to their

final values. We then want to perform c copy operations. At the end, b is equal to the `max_state`, and a points to the left of the last copy of the group.

```

20356 \cs_new_protected:Npn \__regex_group_repeat_aux:n #1
20357 {
20358   \__regex_build_transition_right:nNn \__regex_action_free:n
20359   \l__regex_right_state_int \l__regex_max_state_int
20360   \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
20361   \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
20362   \if_int_compare:w \int_eval:n {#1} > 1 \exp_stop_f:
20363     \int_set:Nn \l__regex_internal_c_int
20364     {
20365       ( #1 - 1 )
20366       * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
20367     }
20368     \int_add:Nn \l__regex_right_state_int { \l__regex_internal_c_int }
20369     \int_add:Nn \l__regex_max_state_int { \l__regex_internal_c_int }
20370     \__regex_toks_memcpy:NNn
20371     \l__regex_internal_b_int
20372     \l__regex_internal_a_int
20373     \l__regex_internal_c_int
20374   \fi:
20375 }

```

(End definition for `__regex_group_repeat_aux:n`.)

`__regex_group_repeat:nnN` This function is called to repeat a group at least n times; the case $n = 0$ is very different from $n > 0$. Assume first that $n = 0$. Insert submatch tracking information at the start and end of the group, add a free transition from the right end to the “true” left state a (remember: in this case we had added an extra state before the left state). This forms the loop, which we break away from by adding a free transition from a to a new state.

Now consider the case $n > 0$. Repeat the group n times, chaining various copies with a free transition. Add submatch tracking only to the last copy, then add a free transition from the right end back to the left end of the last copy, either before or after the transition to move on towards the rest of the NFA. This transition can end up before submatch tracking, but that is irrelevant since it only does so when going again through the group, recording new matches. Finally, add a state; we already have a transition pointing to it from `__regex_group_repeat_aux:n`.

```

20376 \cs_new_protected:Npn \__regex_group_repeat:nnN #1#2#3
20377 {
20378   \if_int_compare:w #2 = 0 \exp_stop_f:
20379     \__regex_group_submatches:nnN {#1}
20380     \l__regex_left_state_int \l__regex_right_state_int
20381     \int_set:Nn \l__regex_internal_a_int
20382     { \l__regex_left_state_int - 1 }
20383     \__regex_build_transition_right:nNn \__regex_action_free:n
20384     \l__regex_right_state_int \l__regex_internal_a_int
20385     \__regex_build_new_state:
20386     \if_meaning:w \c_true_bool #3
20387       \__regex_build_transition_left:NNN \__regex_action_free:n
20388       \l__regex_internal_a_int \l__regex_right_state_int
20389     \else:
20390       \__regex_build_transition_right:nNn \__regex_action_free:n
20391       \l__regex_internal_a_int \l__regex_right_state_int

```

```

20392     \fi:
20393 \else:
20394     \__regex_group_repeat_aux:n {#2}
20395     \__regex_group_submatches:nnN {#1}
20396     \l__regex_internal_a_int \l__regex_right_state_int
20397     \if_meaning:w \c_true_bool #3
20398     \__regex_build_transition_right:nNn \__regex_action_free_group:n
20399     \l__regex_right_state_int \l__regex_internal_a_int
20400     \else:
20401     \__regex_build_transition_left:NNN \__regex_action_free_group:n
20402     \l__regex_right_state_int \l__regex_internal_a_int
20403     \fi:
20404     \__regex_build_new_state:
20405 \fi:
20406 }

```

(End definition for `__regex_group_repeat:nnN`.)

`__regex_group_repeat:nnnN` We wish to repeat the group between #2 and #2 + #3 times, with a laziness controlled by #4. We insert submatch tracking up front: in principle, we could avoid recording submatches for the first #2 copies of the group, but that forces us to treat specially the case #2 = 0. Repeat that group with submatch tracking #2 + #3 times (the maximum number of repetitions). Then our goal is to add #3 transitions from the end of the #2-th group, and each subsequent groups, to the end. For a lazy quantifier, we add those transitions to the left states, before submatch tracking. For the greedy case, we add the transitions to the right states, after submatch tracking and the transitions which go on with more repetitions. In the greedy case with #2 = 0, the transition which skips over all copies of the group must be added separately, because its starting state does not follow the normal pattern: we had to add it “by hand” earlier.

```

20407 \cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
20408 {
20409     \__regex_group_submatches:nnN {#1}
20410     \l__regex_left_state_int \l__regex_right_state_int
20411     \__regex_group_repeat_aux:n { #2 + #3 }
20412     \if_meaning:w \c_true_bool #4
20413     \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
20414     \prg_replicate:nn { #3 }
20415     {
20416         \int_sub:Nn \l__regex_left_state_int
20417         { \l__regex_internal_b_int - \l__regex_internal_a_int }
20418         \__regex_build_transition_left:NNN \__regex_action_free:n
20419         \l__regex_left_state_int \l__regex_max_state_int
20420     }
20421     \else:
20422     \prg_replicate:nn { #3 - 1 }
20423     {
20424         \int_sub:Nn \l__regex_right_state_int
20425         { \l__regex_internal_b_int - \l__regex_internal_a_int }
20426         \__regex_build_transition_right:nNn \__regex_action_free:n
20427         \l__regex_right_state_int \l__regex_max_state_int
20428     }
20429     \if_int_compare:w #2 = 0 \exp_stop_f:
20430     \int_set:Nn \l__regex_right_state_int

```

```

20431         { \l__regex_left_state_int - 1 }
20432     \else:
20433         \int_sub:Nn \l__regex_right_state_int
20434         { \l__regex_internal_b_int - \l__regex_internal_a_int }
20435     \fi:
20436     \__regex_build_transition_right:nNn \__regex_action_free:n
20437     \l__regex_right_state_int \l__regex_max_state_int
20438 \fi:
20439 \__regex_build_new_state:
20440 }

```

(End definition for __regex_group_repeat:nnnN.)

37.4.6 Others

```

\__regex_assertion:Nn
\__regex_b_test:
\__regex_anchor:N

```

Usage: __regex_assertion:Nn *<boolean>* {*<test>*}, where the *<test>* is either of the two other functions. Add a free transition to a new state, conditionally to the assertion test. The __regex_b_test: test is used by the \b and \B escape: check if the last character was a word character or not, and do the same to the current character. The boundary-markers of the string are non-word characters for this purpose. Anchors at the start or end of match use __regex_anchor:N, with a position controlled by the integer #1.

```

20441 \cs_new_protected:Npn \__regex_assertion:Nn #1#2
20442 {
20443     \__regex_build_new_state:
20444     \__regex_toks_put_right:Nx \l__regex_left_state_int
20445     {
20446         \exp_not:n {#2}
20447         \__regex_break_point:TF
20448         \bool_if:NF #1 { { } }
20449         {
20450             \__regex_action_free:n
20451             {
20452                 \int_eval:n
20453                 { \l__regex_right_state_int - \l__regex_left_state_int }
20454             }
20455         }
20456         \bool_if:NT #1 { { } }
20457     }
20458 }
20459 \cs_new_protected:Npn \__regex_anchor:N #1
20460 {
20461     \if_int_compare:w #1 = \l__regex_curr_pos_int
20462     \exp_after:wN \__regex_break_true:w
20463     \fi:
20464 }
20465 \cs_new_protected:Npn \__regex_b_test:
20466 {
20467     \group_begin:
20468     \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int
20469     \__regex_prop_w:
20470     \__regex_break_point:TF
20471     { \group_end: \__regex_item_reverse:n \__regex_prop_w: }
20472     { \group_end: \__regex_prop_w: }
20473 }

```


(End definition for `_regex_assertion:Nn`, `_regex_b_test:`, and `_regex_anchor:N`.)

`_regex_command_K:` Change the starting point of the 0-th submatch (full match), and transition to a new state, pretending that this is a fresh thread.

```

20474 \cs_new_protected:Npn \_regex_command_K:
20475 {
20476   \_regex_build_new_state:
20477   \_regex_toks_put_right:Nx \l__regex_left_state_int
20478   {
20479     \_regex_action_submatch:n { 0 < }
20480     \bool_set_true:N \l__regex_fresh_thread_bool
20481     \_regex_action_free:n
20482     {
20483       \int_eval:n
20484       { \l__regex_right_state_int - \l__regex_left_state_int }
20485     }
20486     \bool_set_false:N \l__regex_fresh_thread_bool
20487   }
20488 }

```

(End definition for `_regex_command_K:.`)

37.5 Matching

We search for matches by running all the execution threads through the NFA in parallel, reading one token of the query at each step. The NFA contains “free” transitions to other states, and transitions which “consume” the current token. For free transitions, the instruction at the new state of the NFA is performed immediately. When a transition consumes a character, the new state is appended to a list of “active states”, stored in `\g__regex_thread_state_intarray:` this thread is made active again when the next token is read from the query. At every step (for each token in the query), we unpack that list of active states and the corresponding submatch props, and empty those.

If two paths through the NFA “collide” in the sense that they reach the same state after reading a given token, then they only differ in how they previously matched, and any future execution would be identical for both. (Note that this would be wrong in the presence of back-references.) Hence, we only need to keep one of the two threads: the thread with the highest priority. Our NFA is built in such a way that higher priority actions always come before lower priority actions, which makes things work.

The explanation in the previous paragraph may make us think that we simply need to keep track of which states were visited at a given step: after all, the loop generated when matching `(a?)*` against `a` is broken, isn’t it? No. The group first matches `a`, as it should, then repeats; it attempts to match `a` again but fails; it skips `a`, and finds out that this state has already been seen at this position in the query: the match stops. The capturing group is (wrongly) `a`. What went wrong is that a thread collided with itself, and the later version, which has gone through the group one more times with an empty match, should have a higher priority than not going through the group.

We solve this by distinguishing “normal” free transitions `_regex_action_free:n` from transitions `_regex_action_free_group:n` which go back to the start of the group. The former keeps threads unless they have been visited by a “completed” thread, while the latter kind of transition also prevents going back to a state visited by the current thread.

37.5.1 Variables used when matching

`\l__regex_min_pos_int` The tokens in the query are indexed from `min_pos` for the first to `max_pos - 1` for the last, and their information is stored in several arrays and `\toks` registers with those numbers.

`\l__regex_max_pos_int` We don't start from 0 because the `\toks` registers with low numbers are used to hold the states of the NFA. We match without backtracking, keeping all threads in lockstep

`\l__regex_curr_pos_int` at the `current_pos` in the query. The starting point of the current match attempt is

`\l__regex_start_pos_int` `start_pos`, and `success_pos`, updated whenever a thread succeeds, is used as the next

`\l__regex_success_pos_int` starting position.

```
20489 \int_new:N \l__regex_min_pos_int
20490 \int_new:N \l__regex_max_pos_int
20491 \int_new:N \l__regex_curr_pos_int
20492 \int_new:N \l__regex_start_pos_int
20493 \int_new:N \l__regex_success_pos_int
```

(End definition for `\l__regex_min_pos_int` and others.)

`\l__regex_curr_char_int` The character and category codes of the token at the current position; the character code

`\l__regex_curr_catcode_int` of the token at the previous position; and the character code of the result of changing the

`\l__regex_last_char_int` case of the current token (`A-Z ↔ a-z`). This last integer is only computed when necessary, and is otherwise `\c_max_int`. The `current_char` variable is also used in various other

`\l__regex_case_changed_char_int` phases to hold a character code.

```
20494 \int_new:N \l__regex_curr_char_int
20495 \int_new:N \l__regex_curr_catcode_int
20496 \int_new:N \l__regex_last_char_int
20497 \int_new:N \l__regex_case_changed_char_int
```

(End definition for `\l__regex_curr_char_int` and others.)

`\l__regex_curr_state_int` For every character in the token list, each of the active states is considered in turn. The variable `\l__regex_curr_state_int` holds the state of the NFA which is currently considered: transitions are then given as shifts relative to the current state.

```
20498 \int_new:N \l__regex_curr_state_int
```

(End definition for `\l__regex_curr_state_int`.)

`\l__regex_curr_submatches_prop` The submatches for the thread which is currently active are stored in the `current_submatches` property list variable. This property list is stored by `__regex_action_cost:n` into the `\toks` register for the target state of the transition, to be retrieved when

`\l__regex_success_submatches_prop` matching at the next position. When a thread succeeds, this property list is copied to `\l__regex_success_submatches_prop`: only the last successful thread remains there.

```
20499 \prop_new:N \l__regex_curr_submatches_prop
20500 \prop_new:N \l__regex_success_submatches_prop
```

(End definition for `\l__regex_curr_submatches_prop` and `\l__regex_success_submatches_prop`.)

`\l__regex_step_int` This integer, always even, is increased every time a character in the query is read, and not reset when doing multiple matches. We store in `\g__regex_state_active_intarray` the last step in which each `<state>` in the NFA was encountered. This lets us break infinite loops by not visiting the same state twice in the same step. In fact, the step we store is equal to `step` when we have started performing the operations of `\toks<state>`, but not finished yet. However, once we finish, we store `step + 1` in `\g__regex_state_active_intarray`. This is needed to track submatches properly (see building phase).

The **step** is also used to attach each set of submatch information to a given iteration (and automatically discard it when it corresponds to a past step).

```
20501 \int_new:N \l__regex_step_int
```

(End definition for `\l__regex_step_int`.)

```
\l__regex_min_active_int
\l__regex_max_active_int
```

All the currently active threads are kept in order of precedence in `\g__regex_thread_state_intarray`, and the corresponding submatches in the `\toks`. For our purposes, those serve as an array, indexed from `min_active` (inclusive) to `max_active` (excluded). At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and `max_active` is reset to `min_active`, effectively clearing the array.

```
20502 \int_new:N \l__regex_min_active_int
```

```
20503 \int_new:N \l__regex_max_active_int
```

(End definition for `\l__regex_min_active_int` and `\l__regex_max_active_int`.)

```
\g__regex_state_active_intarray
\g__regex_thread_state_intarray
```

`\g__regex_state_active_intarray` stores the last *step* in which each *state* was active. `\g__regex_thread_state_intarray` stores threads to be considered in the next step, more precisely the states in which these threads are.

```
20504 \intarray_new:Nn \g__regex_state_active_intarray { 65536 }
```

```
20505 \intarray_new:Nn \g__regex_thread_state_intarray { 65536 }
```

(End definition for `\g__regex_state_active_intarray` and `\g__regex_thread_state_intarray`.)

```
\l__regex_every_match_tl
```

Every time a match is found, this token list is used. For single matching, the token list is empty. For multiple matching, the token list is set to repeat the matching, after performing some operation which depends on the user function. See `__regex_single_match:` and `__regex_multi_match:n`.

```
20506 \tl_new:N \l__regex_every_match_tl
```

(End definition for `\l__regex_every_match_tl`.)

```
\l__regex_fresh_thread_bool
\l__regex_empty_success_bool
\__regex_if_two_empty_matches:F
```

When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When an empty token list is matched, the next successful match of the same empty token list is suppressed. We detect empty matches by setting `\l__regex_fresh_thread_bool` to `true` for threads which directly come from the start of the regex or from the `\K` command, and testing that boolean whenever a thread succeeds. The function `__regex_if_two_empty_matches:F` is redefined at every match attempt, depending on whether the previous match was empty or not: if it was, then the function must cancel a purported success if it is empty and at the same spot as the previous match; otherwise, we definitely don't have two identical empty matches, so the function is `\use:n`.

```
20507 \bool_new:N \l__regex_fresh_thread_bool
```

```
20508 \bool_new:N \l__regex_empty_success_bool
```

```
20509 \cs_new_eq:NN \__regex_if_two_empty_matches:F \use:n
```

(End definition for `\l__regex_fresh_thread_bool`, `\l__regex_empty_success_bool`, and `__regex_if_two_empty_matches:F`.)

`\g__regex_success_bool` The boolean `\l__regex_match_success_bool` is true if the current match attempt was successful, and `\g__regex_success_bool` is true if there was at least one successful match. This is the only global variable in this whole module, but we would need it to be local when matching a control sequence with `\c{...}`. This is done by saving the global variable into `\l__regex_saved_success_bool`, which is local, hence not affected by the changes due to inner regex functions.

```

20510 \bool_new:N \g__regex_success_bool
20511 \bool_new:N \l__regex_saved_success_bool
20512 \bool_new:N \l__regex_match_success_bool

```

(End definition for `\g__regex_success_bool`, `\l__regex_saved_success_bool`, and `\l__regex_match_success_bool`.)

37.5.2 Matching: framework

`__regex_match:n` First store the query into `\toks` registers and arrays (see `__regex_query_set:nnn`).
`__regex_match_init:` Then initialize the variables that should be set once for each user function (even for multiple matches). Namely, the overall matching is not yet successful; none of the states should be marked as visited (`\g__regex_state_active_intarray`), and we start at step 0; we pretend that there was a previous match ending at the start of the query, which was not empty (to avoid smothering an empty match at the start). Once all this is set up, we are ready for the ride. Find the first match.

```

20513 \__kernel_patch:nnNNpn
20514 {
20515   \__regex_trace_push:nnN { regex } { 1 } \__regex_match:n
20516   \__regex_trace:nnx { regex } { 1 } { analyzing-query-token-list }
20517 }
20518 { \__regex_trace_pop:nnN { regex } { 1 } \__regex_match:n }
20519 \cs_new_protected:Npn \__regex_match:n #1
20520 {
20521   \int_zero:N \l__regex_balance_int
20522   \int_set:Nn \l__regex_curr_pos_int { 2 * \l__regex_max_state_int }
20523   \__regex_query_set:nnn { } { -1 } { -2 }
20524   \int_set_eq:NN \l__regex_min_pos_int \l__regex_curr_pos_int
20525   \tl_analysis_map_inline:nn {#1}
20526   { \__regex_query_set:nnn {##1} {"##3"} {##2} }
20527   \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
20528   \__regex_query_set:nnn { } { -1 } { -2 }
20529   \__regex_match_init:
20530   \__regex_match_once:
20531 }
20532 \__kernel_patch:nnNNpn
20533 { \__regex_trace:nnx { regex } { 1 } { initializing } }
20534 { }
20535 \cs_new_protected:Npn \__regex_match_init:
20536 {
20537   \bool_gset_false:N \g__regex_success_bool
20538   \int_step_inline:nnn
20539     \l__regex_min_state_int { \l__regex_max_state_int - 1 }
20540     {
20541       \__kernel_intarray_gset:Nnn
20542       \g__regex_state_active_intarray {##1} { 1 }
20543     }

```

```

20544     \int_set_eq:NN \l__regex_min_active_int \l__regex_max_state_int
20545     \int_zero:N \l__regex_step_int
20546     \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
20547     \int_set:Nn \l__regex_min_submatch_int
20548         { 2 * \l__regex_max_state_int }
20549     \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
20550     \bool_set_false:N \l__regex_empty_success_bool
20551 }

```

(End definition for `__regex_match:n` and `__regex_match_init:.`)

`__regex_match_once:` This function finds one match, then does some action defined by the `every_match` token list, which may recursively call `__regex_match_once:.` First initialize some variables: set the conditional which detects identical empty matches; this match attempt starts at the previous `success_pos`, is not yet successful, and has no submatches yet; clear the array of active threads, and put the starting state 0 in it. We are then almost ready to read our first token in the query, but we actually start one position earlier than the start, and `get` that token, to set `last_char` properly for word boundaries. Then call `__regex_match_loop:`, which runs through the query until the end or until a successful match breaks early.

```

20552 \cs_new_protected:Npn \__regex_match_once:
20553 {
20554     \if_meaning:w \c_true_bool \l__regex_empty_success_bool
20555         \cs_set:Npn \__regex_if_two_empty_matches:F
20556         {
20557             \int_compare:nNnF
20558                 \l__regex_start_pos_int = \l__regex_curr_pos_int
20559         }
20560     \else:
20561         \cs_set_eq:NN \__regex_if_two_empty_matches:F \use:n
20562     \fi:
20563     \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
20564     \bool_set_false:N \l__regex_match_success_bool
20565     \prop_clear:N \l__regex_curr_submatches_prop
20566     \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
20567     \__regex_store_state:n { \l__regex_min_state_int }
20568     \int_set:Nn \l__regex_curr_pos_int
20569         { \l__regex_start_pos_int - 1 }
20570     \__regex_query_get:
20571     \__regex_match_loop:
20572     \l__regex_every_match_tl
20573 }

```

(End definition for `__regex_match_once:.`)

`__regex_single_match:` For a single match, the overall success is determined by whether the only match attempt is a success. When doing multiple matches, the overall matching is successful as soon as any match succeeds. Perform the action #1, then find the next match.

```

20574 \cs_new_protected:Npn \__regex_single_match:
20575 {
20576     \tl_set:Nn \l__regex_every_match_tl
20577     {
20578         \bool_gset_eq:NN
20579             \g__regex_success_bool

```

```

20580         \l__regex_match_success_bool
20581     }
20582 }
20583 \cs_new_protected:Npn \__regex_multi_match:n #1
20584 {
20585     \tl_set:Nn \l__regex_every_match_tl
20586     {
20587         \if_meaning:w \c_true_bool \l__regex_match_success_bool
20588             \bool_gset_true:N \g__regex_success_bool
20589             #1
20590         \exp_after:wN \__regex_match_once:
20591     \fi:
20592 }
20593 }

```

(End definition for __regex_single_match: and __regex_multi_match:n.)

__regex_match_loop: At each new position, set some variables and get the new character and category from the query. Then unpack the array of active threads, and clear it by resetting its length (max_active). This results in a sequence of __regex_use_state_and_submatches:nn {<state>} {<prop>}, and we consider those states one by one in order. As soon as a thread succeeds, exit the step, and, if there are threads to consider at the next position, and we have not reached the end of the string, repeat the loop. Otherwise, the last thread that succeeded is what __regex_match_once: matches. We explain the fresh_thread business when describing __regex_action_wildcard:.

```

20594 \cs_new_protected:Npn \__regex_match_loop:
20595 {
20596     \int_add:Nn \l__regex_step_int { 2 }
20597     \int_incr:N \l__regex_curr_pos_int
20598     \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
20599     \int_set_eq:NN \l__regex_case_changed_char_int \c_max_int
20600     \__regex_query_get:
20601     \use:x
20602     {
20603         \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
20604         \int_step_function:nnN
20605             { \l__regex_min_active_int }
20606             { \l__regex_max_active_int - 1 }
20607         \__regex_match_one_active:n
20608     }
20609     \prg_break_point:
20610     \bool_set_false:N \l__regex_fresh_thread_bool
20611     \if_int_compare:w \l__regex_max_active_int > \l__regex_min_active_int
20612         \if_int_compare:w \l__regex_curr_pos_int < \l__regex_max_pos_int
20613             \exp_after:wN \exp_after:wN \exp_after:wN \__regex_match_loop:
20614         \fi:
20615     \fi:
20616 }
20617 \cs_new:Npn \__regex_match_one_active:n #1
20618 {
20619     \__regex_use_state_and_submatches:nn
20620     { \__kernel_intarray_item:Nn \g__regex_thread_state_intarray {#1} }
20621     { \__regex_toks_use:w #1 }
20622 }

```

(End definition for `_regex_match_loop:` and `_regex_match_one_active:n`.)

`_regex_query_set:nnn` The arguments are: tokens that `o` and `x` expand to one token of the query, the catcode, and the character code. Store those, and the current brace balance (used later to check for overall brace balance) in a `\toks` register and some arrays, then update the `balance`.

```

20623 \cs_new_protected:Npn \_regex_query_set:nnn #1#2#3
20624 {
20625   \_kernel_intarray_gset:Nnn \g__regex_charcode_intarray
20626     { \l__regex_curr_pos_int } {#3}
20627   \_kernel_intarray_gset:Nnn \g__regex_catcode_intarray
20628     { \l__regex_curr_pos_int } {#2}
20629   \_kernel_intarray_gset:Nnn \g__regex_balance_intarray
20630     { \l__regex_curr_pos_int } { \l__regex_balance_int }
20631   \_regex_toks_set:Nn \l__regex_curr_pos_int {#1}
20632   \int_incr:N \l__regex_curr_pos_int
20633   \if_case:w #2 \exp_stop_f:
20634     \or: \int_incr:N \l__regex_balance_int
20635     \or: \int_decr:N \l__regex_balance_int
20636   \fi:
20637 }

```

(End definition for `_regex_query_set:nnn`.)

`_regex_query_get:` Extract the current character and category codes at the current position from the appropriate arrays.

```

20638 \cs_new_protected:Npn \_regex_query_get:
20639 {
20640   \l__regex_curr_char_int
20641     = \_kernel_intarray_item:Nn \g__regex_charcode_intarray
20642       { \l__regex_curr_pos_int } \scan_stop:
20643   \l__regex_curr_catcode_int
20644     = \_kernel_intarray_item:Nn \g__regex_catcode_intarray
20645       { \l__regex_curr_pos_int } \scan_stop:
20646 }

```

(End definition for `_regex_query_get:.`)

37.5.3 Using states of the nfa

`_regex_use_state:` Use the current NFA instruction. The state is initially marked as belonging to the current **step**: this allows normal free transition to repeat, but group-repeating transitions won't. Once we are done exploring all the branches it spawned, the state is marked as **step + 1**: any thread hitting it at that point will be terminated.

```

20647 \_kernel_patch:nnNNpn
20648 {
20649   \_regex_trace:nnx { regex } { 2 }
20650   { state~\int_use:N \l__regex_curr_state_int }
20651 }
20652 { }
20653 \cs_new_protected:Npn \_regex_use_state:
20654 {
20655   \_kernel_intarray_gset:Nnn \g__regex_state_active_intarray
20656     { \l__regex_curr_state_int } { \l__regex_step_int }
20657   \_regex_toks_use:w \l__regex_curr_state_int

```

```

20658     \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
20659     { \l__regex_curr_state_int }
20660     { \int_eval:n { \l__regex_step_int + 1 } }
20661 }

```

(End definition for __regex_use_state:.)

__regex_use_state_and_submatches:nn This function is called as one item in the array of active threads after that array has been unpacked for a new step. Update the `current_state` and `current_submatches` and use the state if it has not yet been encountered at this step.

```

20662 \cs_new_protected:Npn \__regex_use_state_and_submatches:nn #1 #2
20663 {
20664     \int_set:Nn \l__regex_curr_state_int {#1}
20665     \if_int_compare:w
20666         \__kernel_intarray_item:Nn \g__regex_state_active_intarray
20667         { \l__regex_curr_state_int }
20668         < \l__regex_step_int
20669         \tl_set:Nn \l__regex_curr_submatches_prop {#2}
20670         \exp_after:wN \__regex_use_state:
20671     \fi:
20672     \scan_stop:
20673 }

```

(End definition for __regex_use_state_and_submatches:nn.)

37.5.4 Actions when matching

__regex_action_start_wildcard: For an unanchored match, state 0 has a free transition to the next and a costly one to itself, to repeat at the next position. To catch repeated identical empty matches, we need to know if a successful thread corresponds to an empty match. The instruction resetting `\l__regex_fresh_thread_bool` may be skipped by a successful thread, hence we had to add it to `__regex_match_loop`: too.

```

20674 \cs_new_protected:Npn \__regex_action_start_wildcard:
20675 {
20676     \bool_set_true:N \l__regex_fresh_thread_bool
20677     \__regex_action_free:n {1}
20678     \bool_set_false:N \l__regex_fresh_thread_bool
20679     \__regex_action_cost:n {0}
20680 }

```

(End definition for __regex_action_start_wildcard:.)

__regex_action_free:n These functions copy a thread after checking that the NFA state has not already been used at this position. If not, store submatches in the new state, and insert the instructions for that state in the input stream. Then restore the old value of `\l__regex_curr_state_int` and of the current submatches. The two types of free transitions differ by how they test that the state has not been encountered yet: the `group` version is stricter, and will not use a state if it was used earlier in the current thread, hence forcefully breaking the loop, while the “normal” version will revisit a state even within the thread itself.

```

20681 \cs_new_protected:Npn \__regex_action_free:n
20682 { \__regex_action_free_aux:nn { > \l__regex_step_int } }
20683 \cs_new_protected:Npn \__regex_action_free_group:n
20684 { \__regex_action_free_aux:nn { < \l__regex_step_int } }

```



```

20685 \cs_new_protected:Npn \__regex_action_free_aux:nn #1#2
20686 {
20687   \use:x
20688   {
20689     \int_add:Nn \l__regex_curr_state_int {#2}
20690     \exp_not:n
20691     {
20692       \if_int_compare:w
20693         \__kernel_intarray_item:Nn \g__regex_state_active_intarray
20694         { \l__regex_curr_state_int }
20695         #1
20696         \exp_after:wN \__regex_use_state:
20697         \fi:
20698       }
20699       \int_set:Nn \l__regex_curr_state_int
20700       { \int_use:N \l__regex_curr_state_int }
20701       \tl_set:Nn \exp_not:N \l__regex_curr_submatches_prop
20702       { \exp_not:o \l__regex_curr_submatches_prop }
20703     }
20704   }

```

(End definition for __regex_action_free:n, __regex_action_free_group:n, and __regex_action_free_aux:nn.)

__regex_action_cost:n A transition which consumes the current character and shifts the state by #1. The resulting state is stored in the appropriate array for use at the next position, and we also store the current submatches.

```

20705 \cs_new_protected:Npn \__regex_action_cost:n #1
20706 {
20707   \exp_args:Nx \__regex_store_state:n
20708   { \int_eval:n { \l__regex_curr_state_int + #1 } }
20709 }

```

(End definition for __regex_action_cost:n.)

__regex_store_state:n Put the given state in \g__regex_thread_state_intarray, and increment the length of the array. Also store the current submatch in the appropriate \toks.

```

20710 \cs_new_protected:Npn \__regex_store_state:n #1
20711 {
20712   \__regex_store_submatches:
20713   \__kernel_intarray_gset:Nnn \g__regex_thread_state_intarray
20714   { \l__regex_max_active_int } {#1}
20715   \int_incr:N \l__regex_max_active_int
20716 }
20717 \cs_new_protected:Npn \__regex_store_submatches:
20718 {
20719   \__regex_toks_set:No \l__regex_max_active_int
20720   { \l__regex_curr_submatches_prop }
20721 }

```

(End definition for __regex_store_state:n and __regex_store_submatches:.)

__regex_disable_submatches: Some user functions don't require tracking submatches. We get a performance improvement by simply defining the relevant functions to remove their argument and do nothing with it.

```

20722 \cs_new_protected:Npn \__regex_disable_submatches:
20723 {
20724   \cs_set_protected:Npn \__regex_store_submatches: { }
20725   \cs_set_protected:Npn \__regex_action_submatch:n ##1 { }
20726 }

```

(End definition for __regex_disable_submatches:.)

__regex_action_submatch:n Update the current submatches with the information from the current position. Maybe a bottleneck.

```

20727 \cs_new_protected:Npn \__regex_action_submatch:n #1
20728 {
20729   \prop_put:Nno \l__regex_curr_submatches_prop {#1}
20730   { \int_use:N \l__regex_curr_pos_int }
20731 }

```

(End definition for __regex_action_submatch:n.)

__regex_action_success: There is a successful match when an execution path reaches the last state in the NFA, unless this marks a second identical empty match. Then mark that there was a successful match; it is empty if it is “fresh”; and we store the current position and submatches. The current step is then interrupted with \prg_break:, and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.

```

20732 \cs_new_protected:Npn \__regex_action_success:
20733 {
20734   \__regex_if_two_empty_matches:F
20735   {
20736     \bool_set_true:N \l__regex_match_success_bool
20737     \bool_set_eq:NN \l__regex_empty_success_bool
20738     \l__regex_fresh_thread_bool
20739     \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
20740     \prop_set_eq:NN \l__regex_success_submatches_prop
20741     \l__regex_curr_submatches_prop
20742     \prg_break:
20743   }
20744 }

```

(End definition for __regex_action_success:.)

37.6 Replacement

37.6.1 Variables and helpers used in replacement

\l__regex_replacement_csnames_int The behaviour of closing braces inside a replacement text depends on whether a sequences \c{ or \u{ has been encountered. The number of “open” such sequences that should be closed by } is stored in \l__regex_replacement_csnames_int, and decreased by 1 by each }.

```

20745 \int_new:N \l__regex_replacement_csnames_int

```

(End definition for \l__regex_replacement_csnames_int.)

`\l_regex_replacement_category_tl` This sequence of letters is used to correctly restore categories in nested constructions such as `\cL(abc\cD(_)d)`.

```
20746 \tl_new:N \l__regex_replacement_category_tl
20747 \seq_new:N \l__regex_replacement_category_seq
```

(End definition for `\l__regex_replacement_category_tl` and `\l__regex_replacement_category_seq`.)

`\l__regex_balance_tl` This token list holds the replacement text for `__regex_replacement_balance_one_match:n` while it is being built incrementally.

```
20748 \tl_new:N \l__regex_balance_tl
```

(End definition for `\l__regex_balance_tl`.)

`__regex_replacement_balance_one_match:n` This expects as an argument the first index of a set of entries in `\g__regex_submatch_begin_intarray` (and related arrays) which hold the submatch information for a given match. It can be used within an integer expression to obtain the brace balance incurred by performing the replacement on that match. This combines the braces lost by removing the match, braces added by all the submatches appearing in the replacement, and braces appearing explicitly in the replacement. Even though it is always redefined before use, we initialize it as for an empty replacement. An important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading + in the actual definition).

```
20749 \cs_new:Npn \__regex_replacement_balance_one_match:n #1
20750 { - \__regex_submatch_balance:n {#1} }
```

(End definition for `__regex_replacement_balance_one_match:n`.)

`__regex_replacement_do_one_match:n` The input is the same as `__regex_replacement_balance_one_match:n`. This function is redefined to expand to the part of the token list from the end of the previous match to a given match, followed by the replacement text. Hence concatenating the result of this function with all possible arguments (one call for each match), as well as the range from the end of the last match to the end of the string, produces the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

```
20751 \cs_new:Npn \__regex_replacement_do_one_match:n #1
20752 {
20753   \__regex_query_range:nn
20754   { \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1} }
20755   { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
20756 }
```

(End definition for `__regex_replacement_do_one_match:n`.)

`__regex_replacement_exp_not:N` This function lets us navigate around the fact that the primitive `\exp_not:n` requires a braced argument. As far as I can tell, it is only needed if the user tries to include in the replacement text a control sequence set equal to a macro parameter character, such as `\c_parameter_token`. Indeed, within an x-expanding assignment, `\exp_not:N #` behaves as a single #, whereas `\exp_not:n {#}` behaves as a doubled ##.

```
20757 \cs_new:Npn \__regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(End definition for `__regex_replacement_exp_not:N`.)

37.6.2 Query and brace balance

`__regex_query_range:nn`
`__regex_query_range_loop:ww`

When it is time to extract submatches from the token list, the various tokens are stored in `\toks` registers numbered from `\l__regex_min_pos_int` inclusive to `\l__regex_max_pos_int` exclusive. The function `__regex_query_range:nn {<min>} {<max>}` unpacks registers from the position `<min>` to the position `<max> - 1` included. Once this is expanded, a second x-expansion results in the actual tokens from the query. That second expansion is only done by user functions at the very end of their operation, after checking (and correcting) the brace balance first.

```

20758 \cs_new:Npn \__regex_query_range:nn #1#2
20759 {
20760   \exp_after:wN \__regex_query_range_loop:ww
20761   \int_value:w \__regex_int_eval:w #1 \exp_after:wN ;
20762   \int_value:w \__regex_int_eval:w #2 ;
20763   \prg_break_point:
20764 }
20765 \cs_new:Npn \__regex_query_range_loop:ww #1 ; #2 ;
20766 {
20767   \if_int_compare:w #1 < #2 \exp_stop_f:
20768   \else:
20769     \exp_after:wN \prg_break:
20770   \fi:
20771   \__regex_toks_use:w #1 \exp_stop_f:
20772   \exp_after:wN \__regex_query_range_loop:ww
20773   \int_value:w \__regex_int_eval:w #1 + 1 ; #2 ;
20774 }
```

(End definition for `__regex_query_range:nn` and `__regex_query_range_loop:ww`.)

`__regex_query_submatch:n`

Find the start and end positions for a given submatch (of a given match).

```

20775 \cs_new:Npn \__regex_query_submatch:n #1
20776 {
20777   \__regex_query_range:nn
20778   { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
20779   { \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray {#1} }
20780 }
```

(End definition for `__regex_query_submatch:n`.)

`__regex_submatch_balance:n`

Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the brace balances at the `<max pos>` and `<min pos>`. These two positions are found in the corresponding “submatch” arrays.

```

20781 \cs_new_protected:Npn \__regex_submatch_balance:n #1
20782 {
20783   \int_eval:n
20784   {
20785     \int_compare:nNnTF
20786     {
20787       \__kernel_intarray_item:Nn
20788       \g__regex_submatch_end_intarray {#1}
20789     }
20790     = 0
```

```

20791     { 0 }
20792     {
20793         \__kernel_intarray_item:Nn \g__regex_balance_intarray
20794         {
20795             \__kernel_intarray_item:Nn
20796             \g__regex_submatch_end_intarray {#1}
20797         }
20798     }
20799     -
20800     \int_compare:nNnTF
20801     {
20802         \__kernel_intarray_item:Nn
20803         \g__regex_submatch_begin_intarray {#1}
20804     }
20805     = 0
20806     { 0 }
20807     {
20808         \__kernel_intarray_item:Nn \g__regex_balance_intarray
20809         {
20810             \__kernel_intarray_item:Nn
20811             \g__regex_submatch_begin_intarray {#1}
20812         }
20813     }
20814 }
20815 }

```

(End definition for __regex_submatch_balance:n.)

37.6.3 Framework

```

\__regex_replacement:n
\__regex_replacement_aux:n

```

The replacement text is built incrementally. We keep track in \l__regex_balance_int of the balance of explicit begin- and end-group tokens and we store in \l__regex_balance_tl some code to compute the brace balance from submatches (see its description). Detect unescaped right braces, and escaped characters, with trailing \prg_do_nothing: because some of the later function look-ahead. Once the whole replacement text has been parsed, make sure that there is no open csname. Finally, define the balance_one_match and do_one_match functions.

```

20816 \__kernel_patch:nnNNpn
20817 { \__regex_trace_push:nnN { regex } { 1 } \__regex_replacement:n }
20818 { \__regex_trace_pop:nnN { regex } { 1 } \__regex_replacement:n }
20819 \cs_new_protected:Npn \__regex_replacement:n #1
20820 {
20821     \group_begin:
20822     \tl_build_begin:N \l__regex_build_tl
20823     \int_zero:N \l__regex_balance_int
20824     \tl_clear:N \l__regex_balance_tl
20825     \__regex_escape_use:nnnn
20826     {
20827         \if_charcode:w \c_right_brace_str ##1
20828             \__regex_replacement_rbrace:N
20829         \else:
20830             \__regex_replacement_normal:n
20831         \fi:
20832         ##1

```

```

20833     }
20834     { \_regex_replacement_escaped:N ##1 }
20835     { \_regex_replacement_normal:n ##1 }
20836     {#1}
20837     \prg_do_nothing: \prg_do_nothing:
20838     \if_int_compare:w \l__regex_replacement_csnames_int > 0 \exp_stop_f:
20839     \_kernel_msg_error:nxx { kernel } { replacement-missing-rbrace }
20840     { \int_use:N \l__regex_replacement_csnames_int }
20841     \tl_build_put_right:Nx \l__regex_build_tl
20842     { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
20843     \fi:
20844     \seq_if_empty:NF \l__regex_replacement_category_seq
20845     {
20846         \_kernel_msg_error:nxx { kernel } { replacement-missing-rparen }
20847         { \seq_count:N \l__regex_replacement_category_seq }
20848         \seq_clear:N \l__regex_replacement_category_seq
20849     }
20850     \cs_gset:Npx \_regex_replacement_balance_one_match:n ##1
20851     {
20852         + \int_use:N \l__regex_balance_int
20853         \l__regex_balance_tl
20854         - \_regex_submatch_balance:n {##1}
20855     }
20856     \tl_build_end:N \l__regex_build_tl
20857     \exp_args:NNo
20858     \group_end:
20859     \_regex_replacement_aux:n \l__regex_build_tl
20860 }
20861 \cs_new_protected:Npn \_regex_replacement_aux:n #1
20862 {
20863     \cs_set:Npn \_regex_replacement_do_one_match:n ##1
20864     {
20865         \_regex_query_range:nn
20866         {
20867             \_kernel_intarray_item:Nn
20868             \g__regex_submatch_prev_intarray {##1}
20869         }
20870         {
20871             \_kernel_intarray_item:Nn
20872             \g__regex_submatch_begin_intarray {##1}
20873         }
20874         #1
20875     }
20876 }

```

(End definition for `_regex_replacement:n` and `_regex_replacement_aux:n`.)

`_regex_replacement_normal:n` Most characters are simply sent to the output by `\tl_build_put_right:Nn`, unless a particular category code has been requested: then `_regex_replacement_c_A:w` or a similar auxiliary is called. One exception is right parentheses, which restore the category code in place before the group started. Note that the sequence is non-empty there: it contains an empty entry corresponding to the initial value of `\l__regex_replacement_category_tl`.

```

20877 \cs_new_protected:Npn \_regex_replacement_normal:n #1

```

```

20878 {
20879   \tl_if_empty:NTF \l__regex_replacement_category_tl
20880   { \tl_build_put_right:Nn \l__regex_build_tl {#1} }
20881   { % (
20882     \token_if_eq_charcode:NNTF #1 )
20883     {
20884       \seq_pop:NN \l__regex_replacement_category_seq
20885       \l__regex_replacement_category_tl
20886     }
20887     {
20888       \use:c
20889       {
20890         __regex_replacement_c_
20891         \l__regex_replacement_category_tl :w
20892       }
20893       \__regex_replacement_normal:n {#1}
20894     }
20895   }
20896 }

```

(End definition for __regex_replacement_normal:n.)

__regex_replacement_escaped:N As in parsing a regular expression, we use an auxiliary built from #1 if defined. Otherwise, check for escaped digits (standing from submatches from 0 to 9): anything else is a raw character. We use \token_to_str:N to give spaces the right category code.

```

20897 \cs_new_protected:Npn \__regex_replacement_escaped:N #1
20898 {
20899   \cs_if_exist_use:cF { __regex_replacement_#1:w }
20900   {
20901     \if_int_compare:w 1 < 1#1 \exp_stop_f:
20902     \__regex_replacement_put_submatch:n {#1}
20903     \else:
20904       \exp_args:No \__regex_replacement_normal:n
20905       { \token_to_str:N #1 }
20906     \fi:
20907   }
20908 }

```

(End definition for __regex_replacement_escaped:N.)

37.6.4 Submatches

__regex_replacement_put_submatch:n Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a \c{...} or \u{...} construction, it must be taken into account in the brace balance. Later on, ##1 will be replaced by a pointer to the 0-th submatch for a given match. There is an \exp_not:N here as at the point-of-use of \l__regex_balance_tl there is an x-type expansion which is needed to get ##1 in correctly.

```

20909 \cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
20910 {
20911   \if_int_compare:w #1 < \l__regex_capturing_group_int
20912   \tl_build_put_right:Nn \l__regex_build_tl
20913   { \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
20914   \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:

```

```

20915         \tl_put_right:Nn \l__regex_balance_tl
20916         {
20917             + \__regex_submatch_balance:n
20918             { \exp_not:N \int_eval:n { #1 + ##1 } }
20919         }
20920     \fi:
20921 \fi:
20922 }

```

(End definition for __regex_replacement_put_submatch:n.)

__regex_replacement_g:w Grab digits for the \g escape sequence in a primitive assignment to the integer \l__regex_internal_a_int. At the end of the run of digits, check that it ends with a right brace.

```

20923 \cs_new_protected:Npn \__regex_replacement_g:w #1#2
20924 {
20925     \str_if_eq_x:nnTF
20926     { #1#2 }
20927     { \__regex_replacement_normal:n \c_left_brace_str }
20928     { \l__regex_internal_a_int = \__regex_replacement_g_digits:NN }
20929     { \__regex_replacement_error:NNN g #1 #2 }
20930 }
20931 \cs_new:Npn \__regex_replacement_g_digits:NN #1#2
20932 {
20933     \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
20934     {
20935         \if_int_compare:w 1 < 1#2 \exp_stop_f:
20936             #2
20937             \exp_after:wN \use_i:nnn
20938             \exp_after:wN \__regex_replacement_g_digits:NN
20939         \else:
20940             \exp_stop_f:
20941             \exp_after:wN \__regex_replacement_error:NNN
20942             \exp_after:wN g
20943         \fi:
20944     }
20945     {
20946         \exp_stop_f:
20947         \if_meaning:w \__regex_replacement_rbrace:N #1
20948             \exp_args:No \__regex_replacement_put_submatch:n
20949             { \int_use:N \l__regex_internal_a_int }
20950             \exp_after:wN \use_none:nn
20951         \else:
20952             \exp_after:wN \__regex_replacement_error:NNN
20953             \exp_after:wN g
20954         \fi:
20955     }
20956     #1 #2
20957 }

```

(End definition for __regex_replacement_g:w and __regex_replacement_g_digits:NN.)

37.6.5 Csnames in replacement

`__regex_replacement_c:w` `\c` may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with `\u`. Otherwise test whether the category is known; if it is not, complain.

```

20958 \cs_new_protected:Npn \__regex_replacement_c:w #1#2
20959 {
20960   \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
20961   {
20962     \exp_after:wN \token_if_eq_charcode:NNTF \c_left_brace_str #2
20963     { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:N }
20964     {
20965       \cs_if_exist:cTF { \__regex_replacement_c:#2:w }
20966       { \__regex_replacement_cat:NNN #2 }
20967       { \__regex_replacement_error:NNN c #1#2 }
20968     }
20969   }
20970   { \__regex_replacement_error:NNN c #1#2 }
20971 }

```

(End definition for `__regex_replacement_c:w`.)

`__regex_replacement_cu_aux:Nw` Start a control sequence with `\cs:w`, protected from expansion by #1 (either `__regex_replacement_exp_not:N` or `\exp_not:V`), or turned to a string by `\tl_to_str:V` if inside another csname construction `\c` or `\u`. We use `\tl_to_str:V` rather than `\tl_to_str:N` to deal with integers and other registers.

```

20972 \cs_new_protected:Npn \__regex_replacement_cu_aux:Nw #1
20973 {
20974   \if_case:w \l__regex_replacement_csnames_int
20975   \tl_build_put_right:Nn \l__regex_build_tl
20976   { \exp_not:n { \exp_after:wN #1 \cs:w } }
20977   \else:
20978   \tl_build_put_right:Nn \l__regex_build_tl
20979   { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
20980   \fi:
20981   \int_incr:N \l__regex_replacement_csnames_int
20982 }

```

(End definition for `__regex_replacement_cu_aux:Nw`.)

`__regex_replacement_u:w` Check that `\u` is followed by a left brace. If so, start a control sequence with `\cs:w`, which is then unpacked either with `\exp_not:V` or `\tl_to_str:V` depending on the current context.

```

20983 \cs_new_protected:Npn \__regex_replacement_u:w #1#2
20984 {
20985   \str_if_eq_x:nnTF
20986   { #1#2 }
20987   { \__regex_replacement_normal:n \c_left_brace_str }
20988   { \__regex_replacement_cu_aux:Nw \exp_not:V }
20989   { \__regex_replacement_error:NNN u #1#2 }
20990 }

```

(End definition for `__regex_replacement_u:w`.)

`_regex_replacement_rbrace:N` Within a `\c{...}` or `\u{...}` construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```

20991 \cs_new_protected:Npn \\_regex_replacement_rbrace:N #1
20992 {
20993   \if_int_compare:w \\_regex_replacement_csnames_int > 0 \exp_stop_f:
20994     \tl_build_put_right:Nn \\_regex_build_tl { \cs_end: }
20995     \int_decr:N \\_regex_replacement_csnames_int
20996   \else:
20997     \\_regex_replacement_normal:n {#1}
20998   \fi:
20999 }

```

(End definition for `_regex_replacement_rbrace:N`.)

37.6.6 Characters in replacement

`_regex_replacement_cat:NNN` Here, `#1` is a letter among BEMTPUDSLOA and `#2#3` denote the next character. Complain if we reach the end of the replacement or if the construction appears inside `\c{...}` or `\u{...}`, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```

21000 \cs_new_protected:Npn \\_regex_replacement_cat:NNN #1#2#3
21001 {
21002   \token_if_eq_meaning:NNTF \prg_do_nothing: #3
21003   { \\_kernel_msg_error:nn { kernel } { replacement-catcode-end } }
21004   {
21005     \int_compare:nNnTF { \\_regex_replacement_csnames_int } > 0
21006     {
21007       \\_kernel_msg_error:nnnn
21008       { kernel } { replacement-catcode-in-cs } {#1} {#3}
21009       #2 #3
21010     }
21011     {
21012       \str_if_eq:nnTF
21013       { #2#3 }
21014       {
21015         \\_regex_replacement_normal:n ( % )
21016       }
21017       {
21018         \seq_push:NV \\_regex_replacement_category_seq
21019         \\_regex_replacement_category_tl
21020         \tl_set:Nn \\_regex_replacement_category_tl {#1}
21021       }
21022       {
21023         \token_if_eq_meaning:NNT #2 \\_regex_replacement_escaped:N
21024         {
21025           \\_regex_char_if_alphanumeric:NTF #3
21026           {
21027             \\_kernel_msg_error:nnnn
21028             { kernel } { replacement-catcode-escaped }
21029             {#1} {#3}
21030           }
21031           { }
21032         }
21033         \use:c { \\_regex_replacement_c_#1:w } #2 #3

```

```

21034         }
21035     }
21036 }
21037 }

```

(End definition for `_regex_replacement_cat:NNN`.)

We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

```

21038 \group_begin:

```

`_regex_replacement_char:nNN` The only way to produce an arbitrary character-catcode pair is to use the `\lowercase` or `\uppercase` primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: `#3` is the character whose character code to reproduce. We could use `\char_generate:nn` but only for some catcodes (active characters and spaces are not supported).

```

21039 \cs_new_protected:Npn \_regex_replacement_char:nNN #1#2#3
21040 {
21041     \tex_lccode:D 0 = '#3 \scan_stop:
21042     \tex_lowercase:D { \tl_build_put_right:Nn \l__regex_build_tl {#1} }
21043 }

```

(End definition for `_regex_replacement_char:nNN`.)

`_regex_replacement_c_A:w` For an active character, expansion must be avoided, twice because we later do two x-expansions, to unpack `\toks` for the query, and to expand their contents to tokens of the query.

```

21044 \char_set_catcode_active:N \^^@
21045 \cs_new_protected:Npn \_regex_replacement_c_A:w
21046 { \_regex_replacement_char:nNN { \exp_not:n { \exp_not:N \^^@ } } }

```

(End definition for `_regex_replacement_c_A:w`.)

`_regex_replacement_c_B:w` An explicit begin-group token increases the balance, unless within a `\c{...}` or `\u{...}` construction. Add the desired begin-group character, using the standard `\if_false:` trick. We eventually x-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The `\exp_after:wN` is not strictly needed, but is more consistent with l3tl-analysis.

```

21047 \char_set_catcode_group_begin:N \^^@
21048 \cs_new_protected:Npn \_regex_replacement_c_B:w
21049 {
21050     \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:
21051     \int_incr:N \l__regex_balance_int
21052     \fi:
21053     \_regex_replacement_char:nNN
21054     { \exp_not:n { \exp_after:wN \^^@ \if_false: } \fi: } }
21055 }

```

(End definition for `_regex_replacement_c_B:w`.)

`_regex_replacement_c_C:w` This is not quite catcode-related: when the user requests a character with category “control sequence”, the one-character control symbol is returned. As for the active character, we prepare for two x-expansions.

```

21056 \cs_new_protected:Npn \_regex_replacement_c_C:w #1#2
21057 {
21058     \tl_build_put_right:Nn \l__regex_build_tl
21059     { \exp_not:N \exp_not:N \exp_not:c {#2} }
21060 }

```

(End definition for _regex_replacement_c_C:w.)

`_regex_replacement_c_D:w` Subscripts fit the mould: \lowercase the null byte with the correct category.

```

21061 \char_set_catcode_math_subscript:N \^^@
21062 \cs_new_protected:Npn \_regex_replacement_c_D:w
21063 { \_regex_replacement_char:nNN { ^^@ } }

```

(End definition for _regex_replacement_c_D:w.)

`_regex_replacement_c_E:w` Similar to the begin-group case, the second x-expansion produces the bare end-group token.

```

21064 \char_set_catcode_group_end:N \^^@
21065 \cs_new_protected:Npn \_regex_replacement_c_E:w
21066 {
21067     \if_int_compare:w \l__regex_replacement_csnames_int = 0 \exp_stop_f:
21068     \int_decr:N \l__regex_balance_int
21069     \fi:
21070     \_regex_replacement_char:nNN
21071     { \exp_not:n { \if_false: { \fi: ^^@ } } }
21072 }

```

(End definition for _regex_replacement_c_E:w.)

`_regex_replacement_c_L:w` Simply \lowercase a letter null byte to produce an arbitrary letter.

```

21073 \char_set_catcode_letter:N \^^@
21074 \cs_new_protected:Npn \_regex_replacement_c_L:w
21075 { \_regex_replacement_char:nNN { ^^@ } }

```

(End definition for _regex_replacement_c_L:w.)

`_regex_replacement_c_M:w` No surprise here, we lowercase the null math toggle.

```

21076 \char_set_catcode_math_toggle:N \^^@
21077 \cs_new_protected:Npn \_regex_replacement_c_M:w
21078 { \_regex_replacement_char:nNN { ^^@ } }

```

(End definition for _regex_replacement_c_M:w.)

`_regex_replacement_c_O:w` Lowercase an other null byte.

```

21079 \char_set_catcode_other:N \^^@
21080 \cs_new_protected:Npn \_regex_replacement_c_O:w
21081 { \_regex_replacement_char:nNN { ^^@ } }

```

(End definition for _regex_replacement_c_O:w.)

`_regex_replacement_c_P:w` For macro parameters, expansion is a tricky issue. We need to prepare for two x-expansions and passing through various macro definitions. Note that we cannot replace one `\exp_not:n` by doubling the macro parameter characters because this would misbehave if a mischievous user asks for `\c{\cP\#}`, since that macro parameter character would be doubled.

```

21082 \char_set_catcode_parameter:N \^^@
21083 \cs_new_protected:Npn \_regex_replacement_c_P:w
21084 {
21085   \_regex_replacement_char:nNN
21086   { \exp_not:n { \exp_not:n { ^^@@^^@@^^@@ } } }
21087 }

```

(End definition for _regex_replacement_c_P:w.)

`_regex_replacement_c_S:w` Spaces are normalized on input by T_EX to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

```

21088 \cs_new_protected:Npn \_regex_replacement_c_S:w #1#2
21089 {
21090   \if_int_compare:w '#2 = 0 \exp_stop_f:
21091   \_kernel_msg_error:nn { kernel } { replacement-null-space }
21092   \fi:
21093   \tex_lccode:D '\ = '#2 \scan_stop:
21094   \tex_lowercase:D { \tl_build_put_right:Nn \l__regex_build_tl {~} }
21095 }

```

(End definition for _regex_replacement_c_S:w.)

`_regex_replacement_c_T:w` No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don't cause trouble in alignment settings.

```

21096 \char_set_catcode_alignment:N \^^@
21097 \cs_new_protected:Npn \_regex_replacement_c_T:w
21098 { \_regex_replacement_char:nNN { ^^@ } }

```

(End definition for _regex_replacement_c_T:w.)

`_regex_replacement_c_U:w` Simple call to `_regex_replacement_char:nNN` which lowercases the math superscript `^^@`.

```

21099 \char_set_catcode_math_superscript:N \^^@
21100 \cs_new_protected:Npn \_regex_replacement_c_U:w
21101 { \_regex_replacement_char:nNN { ^^@ } }

```

(End definition for _regex_replacement_c_U:w.)

Restore the catcode of the null byte.

```

21102 \group_end:

```

37.6.7 An error

`_regex_replacement_error:NNN` Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```
21103 \cs_new_protected:Npn \_regex_replacement_error:NNN #1#2#3
21104 {
21105     \__kernel_msg_error:nnx { kernel } { replacement-#1 } {#3}
21106     #2 #3
21107 }
```

(End definition for `_regex_replacement_error:NNN`.)

37.7 User functions

`\regex_new:N` Before being assigned a sensible value, a regex variable matches nothing.

```
21108 \cs_new_protected:Npn \regex_new:N #1
21109 { \cs_new_eq:NN #1 \c__regex_no_match_regex }
```

(End definition for `\regex_new:N`. This function is documented on page 209.)

`\l_tmpa_regex` The usual scratch space.

```
\l_tmpb_regex 21110 \regex_new:N \l_tmpa_regex
\g_tmpa_regex 21111 \regex_new:N \l_tmpb_regex
\g_tmpb_regex 21112 \regex_new:N \g_tmpa_regex
21113 \regex_new:N \g_tmpb_regex
```

(End definition for `\l_tmpa_regex` and others. These variables are documented on page 211.)

`\regex_set:Nn` Compile, then store the result in the user variable with the appropriate assignment function.

`\regex_gset:Nn`

`\regex_const:Nn`

```
21114 \cs_new_protected:Npn \regex_set:Nn #1#2
21115 {
21116     \__regex_compile:n {#2}
21117     \tl_set_eq:NN #1 \l__regex_internal_regex
21118 }
21119 \cs_new_protected:Npn \regex_gset:Nn #1#2
21120 {
21121     \__regex_compile:n {#2}
21122     \tl_gset_eq:NN #1 \l__regex_internal_regex
21123 }
21124 \cs_new_protected:Npn \regex_const:Nn #1#2
21125 {
21126     \__regex_compile:n {#2}
21127     \tl_const:Nx #1 { \exp_not:o \l__regex_internal_regex }
21128 }
```

(End definition for `\regex_set:Nn`, `\regex_gset:Nn`, and `\regex_const:Nn`. These functions are documented on page 209.)

`\regex_show:N` User functions: the `n` variant requires compilation first. Then show the variable with some appropriate text. The auxiliary is defined in a different section.

`\regex_show:n`

```
21129 \cs_new_protected:Npn \regex_show:n #1
21130 {
21131     \__regex_compile:n {#1}
21132     \__regex_show:N \l__regex_internal_regex
```

```

21133 \msg_show:nnxxxx { LaTeX / kernel } { show-regex }
21134 { \tl_to_str:n {#1} } { }
21135 { \l__regex_internal_a_tl } { }
21136 }
21137 \cs_new_protected:Npn \regex_show:N #1
21138 {
21139 \__kernel_chk_defined:NT #1
21140 {
21141 \__regex_show:N #1
21142 \msg_show:nnxxxx { LaTeX / kernel } { show-regex }
21143 { } { \token_to_str:N #1 }
21144 { \l__regex_internal_a_tl } { }
21145 }
21146 }

```

(End definition for `\regex_show:N` and `\regex_show:n`. These functions are documented on page 209.)

`\regex_match:nnTF` Those conditionals are based on a common auxiliary defined later. Its first argument
`\regex_match:NnTF` builds the NFA corresponding to the regex, and the second argument is the query token
list. Once we have performed the match, convert the resulting boolean to `\prg_return_`
`true:` or `false`.

```

21147 \prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
21148 {
21149 \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
21150 \__regex_return:
21151 }
21152 \prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
21153 {
21154 \__regex_if_match:nn { \__regex_build:N #1 } {#2}
21155 \__regex_return:
21156 }

```

(End definition for `\regex_match:nnTF` and `\regex_match:NnTF`. These functions are documented on page 209.)

`\regex_count:nnN` Again, use an auxiliary whose first argument builds the NFA.
`\regex_count:NnN`

```

21157 \cs_new_protected:Npn \regex_count:nnN #1
21158 { \__regex_count:nnN { \__regex_build:n {#1} } }
21159 \cs_new_protected:Npn \regex_count:NnN #1
21160 { \__regex_count:nnN { \__regex_build:N #1 } }

```

(End definition for `\regex_count:nnN` and `\regex_count:NnN`. These functions are documented on page 210.)

`\regex_extract_once:nnN` We define here 40 user functions, following a common pattern in terms of `:nnN` auxiliaries,
`\regex_extract_once:nnTF` defined in the coming subsections. The auxiliary is handed `__regex_build:n` or `__`
`\regex_extract_once:NnN` `regex_build:N` with the appropriate regex argument, then all other necessary arguments
`\regex_extract_once:NnTF` (replacement text, token list, *etc.* The conditionals call `__regex_return:` to return
`\regex_extract_all:nnN` either `true` or `false` once matching has been performed.

```

21161 \cs_set_protected:Npn \__regex_tmp:w #1#2#3
21162 {
21163 \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
21164 \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N ##1 } }
21165 \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }

```

`\regex_replace_once:nnN`
`\regex_replace_once:NnN`
`\regex_replace_once:NnTF`
`\regex_replace_all:nnN`
`\regex_replace_all:nnTF`
`\regex_replace_all:NnN`
`\regex_replace_all:NnTF`
`\regex_split:nnN`
`\regex_split:nnTF`
`\regex_split:NnN`
`\regex_split:NnTF`

```

21166     { #1 { \__regex_build:n {##1} } {##2} ##3 \__regex_return: }
21167 \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
21168     { #1 { \__regex_build:N ##1 } {##2} ##3 \__regex_return: }
21169 }
21170 \__regex_tmp:w \__regex_extract_once:nnN
21171 \regex_extract_once:nnN \regex_extract_once:NnN
21172 \__regex_tmp:w \__regex_extract_all:nnN
21173 \regex_extract_all:nnN \regex_extract_all:NnN
21174 \__regex_tmp:w \__regex_replace_once:nnN
21175 \regex_replace_once:nnN \regex_replace_once:NnN
21176 \__regex_tmp:w \__regex_replace_all:nnN
21177 \regex_replace_all:nnN \regex_replace_all:NnN
21178 \__regex_tmp:w \__regex_split:nnN \regex_split:nnN \regex_split:NnN

```

(End definition for `\regex_extract_once:nnNTF` and others. These functions are documented on page 210.)

37.7.1 Variables and helpers for user functions

`\l__regex_match_count_int` The number of matches found so far is stored in `\l__regex_match_count_int`. This is only used in the `\regex_count:nnN` functions.

```
21179 \int_new:N \l__regex_match_count_int
```

(End definition for `\l__regex_match_count_int`.)

`__regex_begin` `__regex_end` Those flags are raised to indicate extra begin-group or end-group tokens when extracting submatches.

```
21180 \flag_new:n { __regex_begin }
21181 \flag_new:n { __regex_end }
```

(End definition for `__regex_begin` and `__regex_end`.)

`\l__regex_min_submatch_int` `\l__regex_submatch_int` `\l__regex_zeroth_submatch_int` The end-points of each submatch are stored in two arrays whose index *submatch* ranges from `\l__regex_min_submatch_int` (inclusive) to `\l__regex_submatch_int` (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labelled starting at `zeroth_submatch`. The entry `\l__regex_zeroth_submatch_int` in `\g__regex_submatch_prev_intarray` holds the position at which that match attempt started: this is used for splitting and replacements.

```
21182 \int_new:N \l__regex_min_submatch_int
21183 \int_new:N \l__regex_submatch_int
21184 \int_new:N \l__regex_zeroth_submatch_int
```

(End definition for `\l__regex_min_submatch_int`, `\l__regex_submatch_int`, and `\l__regex_zeroth_submatch_int`.)

`\g__regex_submatch_prev_intarray` Hold the place where the match attempt begun and the end-points of each submatch.

```

\g__regex_submatch_begin_intarray 21185 \intarray_new:Nn \g__regex_submatch_prev_intarray { 65536 }
\g__regex_submatch_end_intarray    21186 \intarray_new:Nn \g__regex_submatch_begin_intarray { 65536 }
                                   21187 \intarray_new:Nn \g__regex_submatch_end_intarray { 65536 }

```

(End definition for `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray`, and `\g__regex_submatch_end_intarray`.)

`__regex_return:` This function triggers either `\prg_return_false:` or `\prg_return_true:` as appropriate to whether a match was found or not. It is used by all user conditionals.

```

21188 \cs_new_protected:Npn \__regex_return:
21189 {
21190   \if_meaning:w \c_true_bool \g__regex_success_bool
21191     \prg_return_true:
21192   \else:
21193     \prg_return_false:
21194   \fi:
21195 }
```

(End definition for __regex_return:.)

37.7.2 Matching

`__regex_if_match:nn` We don't track submatches, and stop after a single match. Build the NFA with #1, and perform the match on the query #2.

```

21196 \cs_new_protected:Npn \__regex_if_match:nn #1#2
21197 {
21198   \group_begin:
21199     \__regex_disable_submatches:
21200     \__regex_single_match:
21201     #1
21202     \__regex_match:n {#2}
21203   \group_end:
21204 }
```

(End definition for __regex_if_match:nn.)

`__regex_count:nnN` Again, we don't care about submatches. Instead of aborting after the first "longest match" is found, we search for multiple matches, incrementing `\l__regex_match_count_int` every time to record the number of matches. Build the NFA and match. At the end, store the result in the user's variable.

```

21205 \cs_new_protected:Npn \__regex_count:nnN #1#2#3
21206 {
21207   \group_begin:
21208     \__regex_disable_submatches:
21209     \int_zero:N \l__regex_match_count_int
21210     \__regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
21211     #1
21212     \__regex_match:n {#2}
21213     \exp_args:NNNo
21214   \group_end:
21215   \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
21216 }
```

(End definition for __regex_count:nnN.)

37.7.3 Extracting submatches

`__regex_extract_once:nnN` Match once or multiple times. After each match (or after the only match), extract the submatches using `__regex_extract:.` At the end, store the sequence containing all the submatches into the user variable #3 after closing the group.

```

21217 \cs_new_protected:Npn \__regex_extract_once:nnN #1#2#3
21218 {
21219   \group_begin:
21220   \__regex_single_match:
21221   #1
21222   \__regex_match:n {#2}
21223   \__regex_extract:
21224   \__regex_group_end_extract_seq:N #3
21225 }
21226 \cs_new_protected:Npn \__regex_extract_all:nnN #1#2#3
21227 {
21228   \group_begin:
21229   \__regex_multi_match:n { \__regex_extract: }
21230   #1
21231   \__regex_match:n {#2}
21232   \__regex_group_end_extract_seq:N #3
21233 }

```

(End definition for __regex_extract_once:nnN and __regex_extract_all:nnN.)

__regex_split:nnN Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement \l__regex_submatch_int, which controls which matches will be used.

```

21234 \cs_new_protected:Npn \__regex_split:nnN #1#2#3
21235 {
21236   \group_begin:
21237   \__regex_multi_match:n
21238   {
21239     \if_int_compare:w
21240     \l__regex_start_pos_int < \l__regex_success_pos_int
21241     \__regex_extract:
21242     \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
21243     { \l__regex_zeroth_submatch_int } { 0 }
21244     \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
21245     { \l__regex_zeroth_submatch_int }
21246     {
21247       \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray
21248       { \l__regex_zeroth_submatch_int }
21249     }
21250     \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
21251     { \l__regex_zeroth_submatch_int }
21252     { \l__regex_start_pos_int }
21253   \fi:
21254   }
21255   #1
21256   \__regex_match:n {#2}
21257   (assert)\assert_int:n { \l__regex_curr_pos_int = \l__regex_max_pos_int }
21258   \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
21259   { \l__regex_submatch_int } { 0 }
21260   \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray

```

```

21261     { \l__regex_submatch_int }
21262     { \l__regex_max_pos_int }
21263     \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
21264     { \l__regex_submatch_int }
21265     { \l__regex_start_pos_int }
21266     \int_incr:N \l__regex_submatch_int
21267     \if_meaning:w \c_true_bool \l__regex_empty_success_bool
21268     \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
21269     \int_decr:N \l__regex_submatch_int
21270     \fi:
21271     \fi:
21272     \__regex_group_end_extract_seq:N #3
21273   }

```

(End definition for __regex_split:nnN.)

__regex_group_end_extract_seq:N The end-points of submatches are stored as entries of two arrays from \l__regex_min_submatch_int to \l__regex_submatch_int (exclusive). Extract the relevant ranges into \l__regex_internal_a_tl. We detect unbalanced results using the two flags __regex_begin and __regex_end, raised whenever we see too many begin-group or end-group tokens in a submatch.

```

21274 \cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
21275 {
21276     \flag_clear:n { __regex_begin }
21277     \flag_clear:n { __regex_end }
21278     \seq_set_from_function:NnN \l__regex_internal_seq
21279     {
21280         \int_step_function:nnN { \l__regex_min_submatch_int }
21281         { \l__regex_submatch_int - 1 }
21282     }
21283     \__regex_extract_seq_aux:n
21284     \int_compare:nNnF
21285     {
21286         \flag_height:n { __regex_begin } +
21287         \flag_height:n { __regex_end }
21288     }
21289     = 0
21290     {
21291         \__kernel_msg_error:nnxxx { kernel } { result-unbalanced }
21292         { splitting~or~extracting~submatches }
21293         { \flag_height:n { __regex_end } }
21294         { \flag_height:n { __regex_begin } }
21295     }
21296     \seq_set_map:NnN \l__regex_internal_seq \l__regex_internal_seq {##1}
21297     \exp_args:NNNo
21298     \group_end:
21299     \tl_set:Nn #1 { \l__regex_internal_seq }
21300 }

```

(End definition for __regex_group_end_extract_seq:N.)

__regex_extract_seq_aux:n The :n auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

```

21301 \cs_new:Npn \__regex_extract_seq_aux:n #1
21302 {
21303   \exp_after:wN \__regex_extract_seq_aux:ww
21304   \int_value:w \__regex_submatch_balance:n {#1} ; #1;
21305 }
21306 \cs_new:Npn \__regex_extract_seq_aux:ww #1; #2;
21307 {
21308   \if_int_compare:w #1 < 0 \exp_stop_f:
21309     \flag_raise:n { __regex_end }
21310     \prg_replicate:nn {-#1} { \exp_not:n { { \if_false: } \fi: } }
21311   \fi:
21312   \__regex_query_submatch:n {#2}
21313   \if_int_compare:w #1 > 0 \exp_stop_f:
21314     \flag_raise:n { __regex_begin }
21315     \prg_replicate:nn {#1} { \exp_not:n { \if_false: { \fi: } } }
21316   \fi:
21317 }

```

(End definition for __regex_extract_seq_aux:n and __regex_extract_seq_aux:ww.)

__regex_extract: Our task here is to extract from the property list \l__regex_success_submatches_prop the list of end-points of submatches, and store them in appropriate array entries, from \l__regex_extract_b:wn the list of end-points of submatches, and store them in appropriate array entries, from \l__regex_extract_e:wn \l__regex_zeroth_submatch_int upwards. We begin by emptying those entries. Then for each $\langle key \rangle$ - $\langle value \rangle$ pair in the property list update the appropriate entry. This is somewhat a hack: the $\langle key \rangle$ is a non-negative integer followed by < or >, which we use in a comparison to -1. At the end, store the information about the position at which the match attempt started, in \g__regex_submatch_prev_intarray.

```

21318 \cs_new_protected:Npn \__regex_extract:
21319 {
21320   \if_meaning:w \c_true_bool \g__regex_success_bool
21321     \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
21322     \prg_replicate:nn \l__regex_capturing_group_int
21323     {
21324       \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
21325       { \l__regex_submatch_int } { 0 }
21326       \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
21327       { \l__regex_submatch_int } { 0 }
21328       \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
21329       { \l__regex_submatch_int } { 0 }
21330       \int_incr:N \l__regex_submatch_int
21331     }
21332   \prop_map_inline:Nn \l__regex_success_submatches_prop
21333   {
21334     \if_int_compare:w ##1 - 1 \exp_stop_f:
21335       \exp_after:wN \__regex_extract_e:wn \int_value:w
21336     \else:
21337       \exp_after:wN \__regex_extract_b:wn \int_value:w
21338     \fi:
21339     \__regex_int_eval:w \l__regex_zeroth_submatch_int + ##1 {##2}
21340   }
21341   \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
21342   { \l__regex_zeroth_submatch_int } { \l__regex_start_pos_int }
21343   \fi:
21344 }

```

```

21345 \cs_new_protected:Npn \__regex_extract_b:wn #1 < #2
21346 {
21347   \__kernel_intarray_gset:Nnn
21348   \g__regex_submatch_begin_intarray {#1} {#2}
21349 }
21350 \cs_new_protected:Npn \__regex_extract_e:wn #1 > #2
21351 { \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray {#1} {#2} }

```

(End definition for __regex_extract:, __regex_extract_b:wn, and __regex_extract_e:wn.)

37.7.4 Replacement

__regex_replace_once:nnN Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the tokens from the start of the query to the start of the match and the replacement text for this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user's variable after closing the group: this step involves an additional x-expansion, and checks that braces are balanced in the final result.

```

21352 \cs_new_protected:Npn \__regex_replace_once:nnN #1#2#3
21353 {
21354   \group_begin:
21355   \__regex_single_match:
21356   #1
21357   \__regex_replacement:n {#2}
21358   \exp_args:No \__regex_match:n { #3 }
21359   \if_meaning:w \c_false_bool \g__regex_success_bool
21360   \group_end:
21361   \else:
21362     \__regex_extract:
21363     \int_set:Nn \l__regex_balance_int
21364     {
21365       \__regex_replacement_balance_one_match:n
21366       { \l__regex_zeroth_submatch_int }
21367     }
21368     \tl_set:Nx \l__regex_internal_a_tl
21369     {
21370       \__regex_replacement_do_one_match:n
21371       { \l__regex_zeroth_submatch_int }
21372       \__regex_query_range:nn
21373       {
21374         \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
21375         { \l__regex_zeroth_submatch_int }
21376       }
21377       { \l__regex_max_pos_int }
21378     }
21379     \__regex_group_end_replace:N #3
21380   \fi:
21381 }

```

(End definition for __regex_replace_once:nnN.)

`__regex_replace_all:nnN` Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` hold information about submatches of every match in order; each match corresponds to `\l__regex_capturing_group_int` consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

```

21382 \cs_new_protected:Npn \__regex_replace_all:nnN #1#2#3
21383 {
21384   \group_begin:
21385   \__regex_multi_match:n { \__regex_extract: }
21386   #1
21387   \__regex_replacement:n {#2}
21388   \exp_args:No \__regex_match:n {#3}
21389   \int_set:Nn \l__regex_balance_int
21390   {
21391     0
21392     \int_step_function:nnnN
21393     { \l__regex_min_submatch_int }
21394     \l__regex_capturing_group_int
21395     { \l__regex_submatch_int - 1 }
21396     \__regex_replacement_balance_one_match:n
21397   }
21398   \tl_set:Nx \l__regex_internal_a_tl
21399   {
21400     \int_step_function:nnnN
21401     { \l__regex_min_submatch_int }
21402     \l__regex_capturing_group_int
21403     { \l__regex_submatch_int - 1 }
21404     \__regex_replacement_do_one_match:n
21405     \__regex_query_range:nn
21406     \l__regex_start_pos_int \l__regex_max_pos_int
21407   }
21408   \__regex_group_end_replace:N #3
21409 }

```

(End definition for `__regex_replace_all:nnN`.)

`__regex_group_end_replace:N` If the brace balance is not 0, raise an error. Then set the user's variable `#1` to the x-expansion of `\l__regex_internal_a_tl`, adding the appropriate braces to produce a balanced result. And end the group.

```

21410 \cs_new_protected:Npn \__regex_group_end_replace:N #1
21411 {
21412   \if_int_compare:w \l__regex_balance_int = 0 \exp_stop_f:
21413   \else:
21414     \__kernel_msg_error:nnxxx { kernel } { result-unbalanced }
21415     { replacing }
21416     { \int_max:nn { - \l__regex_balance_int } { 0 } }
21417     { \int_max:nn { \l__regex_balance_int } { 0 } }
21418   \fi:
21419   \use:x
21420   {

```

```

21421 \group_end:
21422 \tl_set:Nn \exp_not:N #1
21423 {
21424   \if_int_compare:w \l__regex_balance_int < 0 \exp_stop_f:
21425     \prg_replicate:nn { - \l__regex_balance_int }
21426     { { \if_false: } \fi: }
21427   \fi:
21428   \l__regex_internal_a_tl
21429   \if_int_compare:w \l__regex_balance_int > 0 \exp_stop_f:
21430     \prg_replicate:nn { \l__regex_balance_int }
21431     { \if_false: { \fi: } }
21432   \fi:
21433 }
21434 }
21435 }

```

(End definition for `__regex_group_end_replace:N`.)

37.7.5 Storing and showing compiled patterns

37.8 Messages

Messages for the preparising phase.

```

21436 \use:x
21437 {
21438   \__kernel_msg_new:nnn { kernel } { trailing-backslash }
21439   { Trailing-escape-char~'\iow_char:N\\'~in-regex-or~replacement. }
21440   \__kernel_msg_new:nnn { kernel } { x-missing-rbrace }
21441   {
21442     Missing-brace~'\iow_char:N\}'~in-regex~
21443     '...\iow_char:N\{x\iow_char:N\{...##1'.
21444   }
21445   \__kernel_msg_new:nnn { kernel } { x-overflow }
21446   {
21447     Character-code~##1~too~large~in~
21448     \iow_char:N\{x\iow_char:N\{##2\iow_char:N\}~regex.
21449   }
21450 }

```

Invalid quantifier.

```

21451 \__kernel_msg_new:nnnn { kernel } { invalid-quantifier }
21452 { Braced-quantifier~'#1'~may~not~be~followed~by~'#2'. }
21453 {
21454   The~character~'#2'~is~invalid~in~the~braced~quantifier~'#1'.~
21455   The~only~valid~quantifiers~are~'*',~'?',~'+',~'{<int>}',~
21456   '{<min>}',~and~'{<min>,<max>}',~optionally~followed~by~'?''.
21457 }

```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```

21458 \__kernel_msg_new:nnnn { kernel } { missing-rbrack }
21459 { Missing~right~bracket~inserted~in~regular~expression. }
21460 {
21461   LaTeX~was~given~a~regular~expression~where~a~character~class~
21462   was~started~with~'[',~but~the~matching~']'~is~missing.

```

```

21463 }
21464 \__kernel_msg_new:nnnn { kernel } { missing-rparen }
21465 {
21466     Missing-right~
21467     \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
21468     inserted-in-regular-expression.
21469 }
21470 {
21471     LaTeX-was-given-a-regular-expression-with-\int_eval:n {#1} ~
21472     more-left-parentheses-than-right-parentheses.
21473 }
21474 \__kernel_msg_new:nnnn { kernel } { extra-rparen }
21475 { Extra-right-parenthesis-ignored-in-regular-expression. }
21476 {
21477     LaTeX-came-across-a-closing-parenthesis-when-no-submatch-group-
21478     was-open.~The-parenthesis-will-be-ignored.
21479 }

```

Some escaped alphanumerics are not allowed everywhere.

```

21480 \__kernel_msg_new:nnnn { kernel } { bad-escape }
21481 {
21482     Invalid-escape~'\iow_char:N\\#1'~
21483     \__regex_if_in_cs:TF { within-a-control-sequence. }
21484     {
21485         \__regex_if_in_class:TF
21486         { in-a-character-class. }
21487         { following-a-category-test. }
21488     }
21489 }
21490 {
21491     The-escape-sequence~'\iow_char:N\\#1'~may-not-appear~
21492     \__regex_if_in_cs:TF
21493     {
21494         within-a-control-sequence-test-introduced-by~
21495         '\iow_char:N\\c\iow_char:N{\'.
21496     }
21497     {
21498         \__regex_if_in_class:TF
21499         { within-a-character-class~ }
21500         { following-a-category-test-such-as~'\iow_char:N\\cL'~ }
21501         because-it-does-not-match-exactly-one-character.
21502     }
21503 }

```

Range errors.

```

21504 \__kernel_msg_new:nnnn { kernel } { range-missing-end }
21505 { Invalid-end-point-for-range~'#1-#2'~in-character-class. }
21506 {
21507     The-end-point~'#2'~of-the-range~'#1-#2'~may-not-serve-as-an-
21508     end-point-for-a-range:~alphanumeric-characters-should-not-be-
21509     escaped,~and-non-alphanumeric-characters-should-be-escaped.
21510 }
21511 \__kernel_msg_new:nnnn { kernel } { range-backwards }
21512 { Range~'#1-#2'~out-of-order-in-character-class. }
21513 {

```



```

21514 In-ranges-of-characters~'[x-y]~appearing-in-character-classes,~
21515 the-first-character-code-must-not-be-larger-than-the-second.~
21516 Here,~'#1'~has-character-code~\int_eval:n {'#1},~while~
21517 '#2'~has-character-code~\int_eval:n {'#2}.
21518 }

```

Errors related to \c and \u.

```

21519 \__kernel_msg_new:nnnn { kernel } { c-bad-mode }
21520 { Invalid-nested~'\iow_char:N\\c'~escape-in-regular-expression. }
21521 {
21522 The~'\iow_char:N\\c'~escape-cannot-be-used-within~
21523 a-control-sequence-test~'\iow_char:N\\c{...}'.~
21524 To-combine-several-category-tests,~use~'\iow_char:N\\c[...]'.
21525 }
21526 \__kernel_msg_new:nnnn { kernel } { c-C-invalid }
21527 { '\iow_char:N\\cC'~should-be-followed-by~'.'~or~'(',~not~'#1'. }
21528 {
21529 The~'\iow_char:N\\cC'~construction-restricts-the-next-item-to-be-a~
21530 control-sequence-or-the-next-group-to-be-made-of-control-sequences.~
21531 It-only-makes-sense-to-follow-it-by~'.'~or-by-a-group.
21532 }
21533 \__kernel_msg_new:nnnn { kernel } { c-lparen-in-class }
21534 { Catcode-test-cannot-apply-to-group-in-character-class }
21535 {
21536 Construction-such-as~'\iow_char:N\\cL(abc)'~are-not-allowed-inside-a~
21537 class~'...'~because-classes-do-not-match-multiple-characters-at-once.
21538 }
21539 \__kernel_msg_new:nnnn { kernel } { c-missing-rbrace }
21540 { Missing-right-brace-inserted-for~'\iow_char:N\\c'~escape. }
21541 {
21542 LaTeX-was-given-a-regular-expression-where-a~
21543 '\iow_char:N\\c\iow_char:N{...}'~construction-was-not-ended~
21544 with-a-closing-brace~'\iow_char:N}''.
21545 }
21546 \__kernel_msg_new:nnnn { kernel } { c-missing-rbrack }
21547 { Missing-right-bracket-inserted-for~'\iow_char:N\\c'~escape. }
21548 {
21549 A-construction~'\iow_char:N\\c[...]'~appears-in-a~
21550 regular-expression,~but~the-closing~']'~is-not-present.
21551 }
21552 \__kernel_msg_new:nnnn { kernel } { c-missing-category }
21553 { Invalid-character~'#1'~following~'\iow_char:N\\c'~escape. }
21554 {
21555 In-regular-expressions,~the~'\iow_char:N\\c'~escape-sequence~
21556 may-only-be-followed-by~a-left-brace,~a-left-bracket,~or~a~
21557 capital-letter-representing-a-character-category,~namely~
21558 one-of~'ABCDELMOPSTU'.
21559 }
21560 \__kernel_msg_new:nnnn { kernel } { c-trailing }
21561 { Trailing-category-code-escape~'\iow_char:N\\c'... }
21562 {
21563 A-regular-expression-ends-with~'\iow_char:N\\c'~followed~
21564 by-a-letter.~It-will-be-ignored.
21565 }
21566 \__kernel_msg_new:nnnn { kernel } { u-missing-lbrace }

```

```

21567 { Missing~left~brace~following~'\iow_char:N\\u'~escape. }
21568 {
21569   The~'\iow_char:N\\u'~escape~sequence~must~be~followed~by~
21570   a~brace~group~with~the~name~of~the~variable~to~use.
21571 }
21572 \__kernel_msg_new:nnnn { kernel } { u-missing-rbrace }
21573 { Missing~right~brace~inserted~for~'\iow_char:N\\u'~escape. }
21574 {
21575   LaTeX~
21576   \str_if_eq_x:nnTF { } {#2}
21577   { reached~the~end~of~the~string~ }
21578   { encountered~an~escaped~alphanumeric~character '\iow_char:N\\#2'~ }
21579   when~parsing~the~argument~of~an~
21580   '\iow_char:N\\u\iow_char:N\\{...\\}'~escape.
21581 }

```

Errors when encountering the POSIX syntax [:...:].

```

21582 \__kernel_msg_new:nnnn { kernel } { posix-unsupported }
21583 { POSIX~collating~element~'#1 ~ #1'~not~supported. }
21584 {
21585   The~'[:foo:]'~and~'[=bar=]'~syntaxes~have~a~special~meaning~
21586   in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
21587   Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
21588 }
21589 \__kernel_msg_new:nnnn { kernel } { posix-unknown }
21590 { POSIX~class~'[:#1:]'~unknown. }
21591 {
21592   '[:#1:]'~is~not~among~the~known~POSIX~classes~
21593   '[:alnum:]',~'[:alpha:]',~'[:ascii:]',~'[:blank:]',~
21594   '[:cntrl:]',~'[:digit:]',~'[:graph:]',~'[:lower:]',~
21595   '[:print:]',~'[:punct:]',~'[:space:]',~'[:upper:]',~
21596   '[:word:]',~and~'[:xdigit:]'.
21597 }
21598 \__kernel_msg_new:nnnn { kernel } { posix-missing-close }
21599 { Missing~closing~':'~for~POSIX~class. }
21600 { The~POSIX~syntax~'#1'~must~be~followed~by~':'',~not~'#2'. }

```

In various cases, the result of a `l3regex` operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```

21601 \__kernel_msg_new:nnnn { kernel } { result-unbalanced }
21602 { Missing~brace~inserted~when~#1. }
21603 {
21604   LaTeX~was~asked~to~do~some~regular~expression~operation,~
21605   and~the~resulting~token~list~would~not~have~the~same~number~
21606   of~begin~group~and~end~group~tokens.~Braces~were~inserted:~
21607   #2~left,~#3~right.
21608 }

```

Error message for unknown options.

```

21609 \__kernel_msg_new:nnnn { kernel } { unknown-option }
21610 { Unknown~option~'#1'~for~regular~expressions. }
21611 {
21612   The~only~available~option~is~'case-insensitive',~toggled~by~
21613   '(?i)'~and~'(?-i)'.

```

```

21614 }
21615 \_kernel_msg_new:nnnn { kernel } { special-group-unknown }
21616 { Unknown~special~group~'#1~...'~in~a~regular~expression. }
21617 {
21618   The~only~valid~constructions~starting~with~'(?~are~
21619   '(:~...'~)',~'(?|~...'~)',~'(?i)',~and~'(?-i)'.
21620 }
  Errors in the replacement text.
21621 \_kernel_msg_new:nnnn { kernel } { replacement-c }
21622 { Misused~'\iow_char:N\\c'~command~in~a~replacement~text. }
21623 {
21624   In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
21625   can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~
21626   or~a~brace~group,~not~by~'#1'.
21627 }
21628 \_kernel_msg_new:nnnn { kernel } { replacement-u }
21629 { Misused~'\iow_char:N\\u'~command~in~a~replacement~text. }
21630 {
21631   In~a~replacement~text,~the~'\iow_char:N\\u'~escape~sequence~
21632   must~be~followed~by~a~brace~group~holding~the~name~of~the~
21633   variable~to~use.
21634 }
21635 \_kernel_msg_new:nnnn { kernel } { replacement-g }
21636 {
21637   Missing~brace~for~the~'\iow_char:N\\g'~construction~
21638   in~a~replacement~text.
21639 }
21640 {
21641   In~the~replacement~text~for~a~regular~expression~search,~
21642   submatches~are~represented~either~as~'\iow_char:N \\g{dd..d}',~
21643   or~'\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
21644 }
21645 \_kernel_msg_new:nnnn { kernel } { replacement-catcode-end }
21646 {
21647   Missing~character~for~the~'\iow_char:N\\c<category><character>'~
21648   construction~in~a~replacement~text.
21649 }
21650 {
21651   In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
21652   can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
21653   the~character~category.~Then,~a~character~must~follow.~LaTeX~
21654   reached~the~end~of~the~replacement~when~looking~for~that.
21655 }
21656 \_kernel_msg_new:nnnn { kernel } { replacement-catcode-escaped }
21657 {
21658   Escaped~letter~or~digit~after~category~code~in~replacement~text.
21659 }
21660 {
21661   In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
21662   can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
21663   the~character~category.~Then,~a~character~must~follow,~not~
21664   '\iow_char:N\\#2'.
21665 }
21666 \_kernel_msg_new:nnnn { kernel } { replacement-catcode-in-cs }

```

```

21667 {
21668   Category-code~'\iow_char:N\c#1#3'~ignored~inside~
21669   '\iow_char:N\c\{...\}'~in~a~replacement~text.
21670 }
21671 {
21672   In~a~replacement~text,~the~category~codes~of~the~argument~of~
21673   '\iow_char:N\c\{...\}'~are~ignored~when~building~the~control~
21674   sequence~name.
21675 }
21676 \__kernel_msg_new:nnnn { kernel } { replacement-null-space }
21677 { TeX~cannot~build~a~space~token~with~character~code~0. }
21678 {
21679   You~asked~for~a~character~token~with~category~space,~
21680   and~character~code~0,~for~instance~through~
21681   '\iow_char:N\cS\iow_char:N\x00'.~
21682   This~specific~case~is~impossible~and~will~be~replaced~
21683   by~a~normal~space.
21684 }
21685 \__kernel_msg_new:nnnn { kernel } { replacement-missing-rbrace }
21686 { Missing~right~brace~inserted~in~replacement~text. }
21687 {
21688   There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
21689   missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
21690 }
21691 \__kernel_msg_new:nnnn { kernel } { replacement-missing-rparen }
21692 { Missing~right~parenthesis~inserted~in~replacement~text. }
21693 {
21694   There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
21695   missing~right~
21696   \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
21697 }

```

Used when showing a regex.

```

21698 \__kernel_msg_new:nnn { kernel } { show-regex }
21699 {
21700   >~Compiled~regex~
21701   \tl_if_empty:nTF {#1} { variable~ #2 } { {#1} } :
21702   #3
21703 }

```

`__regex_msg_repeated:nnN` This is not technically a message, but seems related enough to go there. The arguments are: #1 is the minimum number of repetitions; #2 is the number of allowed extra repetitions (−1 for infinite number), and #3 tells us about lazyness.

```

21704 \cs_new:Npn \__regex_msg_repeated:nnN #1#2#3
21705 {
21706   \str_if_eq_x:nnF { #1 #2 } { 1 0 }
21707   {
21708     , ~ repeated ~
21709     \int_case:nnF {#2}
21710     {
21711       { -1 } { #1~or~more~times,~\bool_if:NTF #3 { lazy } { greedy } }
21712       { 0 } { #1~times }
21713     }
21714   }

```

```

21715         between~#1~and~\int_eval:n {#1+#2}~times,~
21716         \bool_if:NTF #3 { lazy } { greedy }
21717     }
21718 }
21719 }

```

(End definition for `_regex_msg_repeated:nnN`.)

37.9 Code for tracing

There is a more extensive implementation of tracing in the `l3trial` package `l3trace`. Function names are a bit different but could be merged.

Here `#1` is the module name (`regex`) and `#2` is typically 1. If the module's current tracing level is less than `#2` show nothing, otherwise write `#3` to the terminal.

```

\_regex_trace_push:nnN
\_regex_trace_pop:nnN
\_regex_trace:nnx
21720 \__kernel_if_debug:TF
21721 {
21722   \cs_new_protected:Npn \_regex_trace_push:nnN #1#2#3
21723   { \_regex_trace:nnx {#1} {#2} { entering~ \token_to_str:N #3 } }
21724   \cs_new_protected:Npn \_regex_trace_pop:nnN #1#2#3
21725   { \_regex_trace:nnx {#1} {#2} { leaving~ \token_to_str:N #3 } }
21726   \cs_new_protected:Npn \_regex_trace:nnx #1#2#3
21727   {
21728     \int_compare:nNnF
21729     { \int_use:c { g\_regex_trace_#1_int } } < {#2}
21730     { \iow_term:x { Trace:~#3 } }
21731   }
21732 }
21733 { }

```

(End definition for `_regex_trace_push:nnN`, `_regex_trace_pop:nnN`, and `_regex_trace:nnx`.)

```

\_g_regex_trace_regex_int
21734 \int_new:N \g\_regex_trace_regex_int

```

(End definition for `\g_regex_trace_regex_int`.)

This function lists the contents of all states of the NFA, stored in `\toks` from 0 to `\l__regex_max_state_int` (excluded).

```

\_regex_trace_states:n
21735 \__kernel_if_debug:TF
21736 {
21737   \cs_new_protected:Npn \_regex_trace_states:n #1
21738   {
21739     \int_step_inline:nnn
21740     \l__regex_min_state_int
21741     { \l__regex_max_state_int - 1 }
21742     {
21743       \_regex_trace:nnx { regex } {#1}
21744       { \iow_char:N \toks ##1 = { \_regex_toks_use:w ##1 } }
21745     }
21746   }
21747 }
21748 { }

```

(End definition for `_regex_trace_states:n`.)

```

21749 </initex | package>

```

38 l3box implementation

```
21750 <*initex | package>
21751 <@@=box>
```

38.1 Support code

`_box_dim_eval:w` Evaluating a dimension expression expandably. The only difference with `\dim_eval:n` is the lack of `\dim_use:N`, to produce an internal dimension rather than expand it into characters.

```
21752 \cs_new_eq:NN \_box_dim_eval:w \etex_dimexpr:D
21753 \_kernel_patch_args:nNnNpn
21754 {
21755   {
21756     \_kernel_chk_expr:nNnN {#1}
21757     \_box_dim_eval:w { } \_box_dim_eval:n
21758   }
21759 }
21760 \cs_new:Npn \_box_dim_eval:n #1
21761 { \_box_dim_eval:w #1 \scan_stop: }
```

(End definition for `_box_dim_eval:w` and `_box_dim_eval:n`.)

38.2 Creating and initialising boxes

The following test files are used for this code: *m3box001.lvt*.

`\box_new:N` Defining a new `<box>` register: remember that box 255 is not generally available.

```
\box_new:c 21762 <*package>
21763 \cs_new_protected:Npn \box_new:N #1
21764 {
21765   \_kernel_chk_if_free_cs:N #1
21766   \cs:w newbox \cs_end: #1
21767 }
21768 </package>
21769 \cs_generate_variant:Nn \box_new:N { c }
```

Clear a `<box>` register.

```
21770 \cs_new_protected:Npn \box_clear:N #1
21771 { \box_set_eq:NN #1 \c_empty_box }
\box_clear:c 21772 \cs_new_protected:Npn \box_gclear:N #1
21773 { \box_gset_eq:NN #1 \c_empty_box }
\box_gclear:N 21774 \cs_generate_variant:Nn \box_clear:N { c }
21775 \cs_generate_variant:Nn \box_gclear:N { c }
```

Clear or new.

```
21776 \cs_new_protected:Npn \box_clear_new:N #1
\box_clear_new:N 21777 { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
21778 \cs_new_protected:Npn \box_gclear_new:N #1
\box_gclear_new:N 21779 { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
21780 \cs_generate_variant:Nn \box_clear_new:N { c }
21781 \cs_generate_variant:Nn \box_gclear_new:N { c }
```

Assigning the contents of a box to be another box.

```

21782 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\box_set_eq:NN 21783 \cs_new_protected:Npn \box_set_eq:NN #1#2
\box_set_eq:cN 21784 { \tex_setbox:D #1 \tex_copy:D #2 }
\box_set_eq:Nc 21785 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
\box_set_eq:cc 21786 \cs_new_protected:Npn \box_gset_eq:NN #1#2
\box_gset_eq:NN 21787 { \tex_global:D \tex_setbox:D #1 \tex_copy:D #2 }
\box_gset_eq:cN 21788 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
\box_gset_eq:Nc 21789 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
\box_gset_eq:cc

```

Assigning the contents of a box to be another box. This clears the second box globally (that's how \TeX does it).

```

\box_set_eq_clear:NN 21790 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\box_set_eq_clear:cN 21791 \cs_new_protected:Npn \box_set_eq_clear:NN #1#2
\box_set_eq_clear:Nc 21792 { \tex_setbox:D #1 \tex_box:D #2 }
\box_set_eq_clear:cc 21793 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
\box_gset_eq_clear:NN 21794 \cs_new_protected:Npn \box_gset_eq_clear:NN #1#2
\box_gset_eq_clear:cN 21795 { \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
\box_gset_eq_clear:Nc 21796 \cs_generate_variant:Nn \box_set_eq_clear:NN { c , Nc , cc }
\box_gset_eq_clear:cc 21797 \cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }

```

Copies of the `cs` functions defined in `l3basics`.

```

21798 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
\box_if_exist_p:N 21799 { TF , T , F , p }
\box_if_exist_p:c 21800 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
\box_if_exist:N $\underline{TF}$  21801 { TF , T , F , p }
\box_if_exist:c $\underline{TF}$ 

```

38.3 Measuring and setting box dimensions

Accessing the height, depth, and width of a $\langle box \rangle$ register.

```

21802 \cs_new_eq:NN \box_ht:N \tex_ht:D
\box_ht:N 21803 \cs_new_eq:NN \box_dp:N \tex_dp:D
\box_ht:c 21804 \cs_new_eq:NN \box_wd:N \tex_wd:D
\box_dp:N 21805 \cs_generate_variant:Nn \box_ht:N { c }
\box_dp:c 21806 \cs_generate_variant:Nn \box_dp:N { c }
\box_wd:N 21807 \cs_generate_variant:Nn \box_wd:N { c }
\box_wd:c

```

Setting the size is easy: all primitive work. These primitives are not expandable, so the derived functions are not either. When debugging, the dimension expression #2 is surrounded by parentheses to catch early termination.

```

\box_set_ht:NN 21808 \cs_new_protected:Npn \box_set_dp:Nn #1#2
\box_set_ht:cn 21809 { \box_dp:N #1 \__box_dim_eval:n {#2} }
\box_set_dp:NN 21810 \cs_new_protected:Npn \box_set_ht:NN #1#2
\box_set_dp:cn 21811 { \box_ht:N #1 \__box_dim_eval:n {#2} }
\box_set_wd:NN 21812 \cs_new_protected:Npn \box_set_wd:Nn #1#2
\box_set_wd:cn 21813 { \box_wd:N #1 \__box_dim_eval:n {#2} }
21814 \cs_generate_variant:Nn \box_set_ht:NN { c }
21815 \cs_generate_variant:Nn \box_set_dp:NN { c }
21816 \cs_generate_variant:Nn \box_set_wd:NN { c }

```

38.4 Using boxes

Using a $\langle box \rangle$. These are just T_EX primitives with meaningful names.

```

21817 \cs_new_eq:NN \box_use_drop:N \tex_box:D
\box_use_drop:N 21818 \cs_new_eq:NN \box_use:N \tex_copy:D
\box_use_drop:c 21819 \cs_generate_variant:Nn \box_use_drop:N { c }
\box_use:N 21820 \cs_generate_variant:Nn \box_use:N { c }
\box_use:c

```

Move box material in different directions. When debugging, the dimension expression #1 is surrounded by parentheses to catch early termination.

```

\box_move_left:nn 21821 \cs_new_protected:Npn \box_move_left:nn #1#2
\box_move_right:nn 21822 { \tex_moveleft:D \_box_dim_eval:n {#1} #2 }
\box_move_up:nn 21823 \cs_new_protected:Npn \box_move_right:nn #1#2
\box_move_down:nn 21824 { \tex_moveright:D \_box_dim_eval:n {#1} #2 }
21825 \cs_new_protected:Npn \box_move_up:nn #1#2
21826 { \tex_raise:D \_box_dim_eval:n {#1} #2 }
21827 \cs_new_protected:Npn \box_move_down:nn #1#2
21828 { \tex_lower:D \_box_dim_eval:n {#1} #2 }

```

38.5 Box conditionals

The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```

21829 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
\if_hbox:N 21830 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
\if_vbox:N 21831 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
\if_box_empty:N

```

```

21832 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
\box_if_horizontal_p:N 21833 { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_horizontal_p:c 21834 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
\box_if_horizontal:N $\underline{TF}$  21835 { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_horizontal:c $\underline{TF}$  21836 \prg_generate_conditional_variant:Nnn \box_if_horizontal:N
\box_if_vertical_p:N 21837 { c } { p , T , F , TF }
\box_if_vertical_p:c 21838 \prg_generate_conditional_variant:Nnn \box_if_vertical:N
\box_if_vertical:N $\underline{TF}$  21839 { c } { p , T , F , TF }
\box_if_vertical:c $\underline{TF}$ 

```

Testing if a $\langle box \rangle$ is empty/void.

```

21840 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
\box_if_empty_p:N 21841 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_empty_p:c 21842 \prg_generate_conditional_variant:Nnn \box_if_empty:N
\box_if_empty:N $\underline{TF}$  21843 { c } { p , T , F , TF }
\box_if_empty:c $\underline{TF}$ 

```

(End definition for $\backslash box_new:N$ and others. These functions are documented on page 215.)

38.6 The last box inserted

```

\box_set_to_last:N Set a box to the previous box.
\box_set_to_last:c 21844 \_kernel_patch:nnNNpn { \_kernel_chk_var_local:N #1 } { }
\box_gset_to_last:N 21845 \cs_new_protected:Npn \box_set_to_last:N #1
\box_gset_to_last:c 21846 { \tex_setbox:D #1 \tex_lastbox:D }
21847 \_kernel_patch:nnNNpn { \_kernel_chk_var_global:N #1 } { }
21848 \cs_new_protected:Npn \box_gset_to_last:N #1
21849 { \tex_global:D \tex_setbox:D #1 \tex_lastbox:D }

```



```

21850 \cs_generate_variant:Nn \box_set_to_last:N { c }
21851 \cs_generate_variant:Nn \box_gset_to_last:N { c }

```

(End definition for `\box_set_to_last:N` and `\box_gset_to_last:N`. These functions are documented on page 218.)

38.7 Constant boxes

`\c_empty_box` A box we never use.

```

21852 \box_new:N \c_empty_box

```

(End definition for `\c_empty_box`. This variable is documented on page 218.)

38.8 Scratch boxes

`\l_tmpa_box` Scratch boxes.

```

\l_tmpb_box 21853 \box_new:N \l_tmpa_box
\l_tmpb_box 21854 \box_new:N \l_tmpb_box
\g_tmpa_box 21855 \box_new:N \g_tmpa_box
\g_tmpb_box 21856 \box_new:N \g_tmpb_box

```

(End definition for `\l_tmpa_box` and others. These variables are documented on page 218.)

38.9 Viewing box contents

TeX's `\showbox` is not really that helpful in many cases, and it is also inconsistent with other L^AT_EX3 show functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

`\box_show:N` Essentially a wrapper around the internal function, but evaluating the breadth and depth arguments now outside the group.

`\box_show:c`

`\box_show:Nnn` 21857 \cs_new_protected:Npn \box_show:N #1

`\box_show:cnn` 21858 { \box_show:Nnn #1 \c_max_int \c_max_int }

21859 \cs_generate_variant:Nn \box_show:N { c }

21860 \cs_new_protected:Npn \box_show:Nnn #1#2#3

21861 { __box_show:NNff 1 #1 { \int_eval:n {#2} } { \int_eval:n {#3} } }

21862 \cs_generate_variant:Nn \box_show:Nnn { c }

(End definition for `\box_show:N` and `\box_show:Nnn`. These functions are documented on page 218.)

`\box_log:N` Getting TeX to write to the log without interruption the run is done by altering the interaction mode. For that, the ε -TeX extensions are needed.

`\box_log:c`

`\box_log:Nnn` 21863 \cs_new_protected:Npn \box_log:N #1

`\box_log:cnn` 21864 { \box_log:Nnn #1 \c_max_int \c_max_int }

`__box_log:nNnn` 21865 \cs_generate_variant:Nn \box_log:N { c }

21866 \cs_new_protected:Npn \box_log:Nnn

21867 { \exp_args:No __box_log:nNnn { \tex_the:D \etex_interactionmode:D } }

21868 \cs_new_protected:Npn __box_log:nNnn #1#2#3#4

21869 {

21870 \int_set:Nn \etex_interactionmode:D { 0 }

21871 __box_show:NNff 0 #2 { \int_eval:n {#3} } { \int_eval:n {#4} }

21872 \int_set:Nn \etex_interactionmode:D {#1}

21873 }

21874 \cs_generate_variant:Nn \box_log:Nnn { c }

(End definition for `\box_log:N`, `\box_log:Nnn`, and `__box_log:nNnn`. These functions are documented on page 219.)

`__box_show:Nnn` The internal auxiliary to actually do the output uses a group to deal with breadth and
`__box_show:NNff` depth values. The `\use:n` here gives better output appearance. Setting `\tracingonline`
and `\errorcontextlines` is used to control what appears in the terminal.

```

21875 \cs_new_protected:Npn \__box_show:Nnn #1#2#3#4
21876 {
21877   \box_if_exist:NTF #2
21878   {
21879     \group_begin:
21880     \int_set:Nn \tex_showboxbreadth:D {#3}
21881     \int_set:Nn \tex_showboxdepth:D {#4}
21882     \int_set:Nn \tex_tracingonline:D {#1}
21883     \int_set:Nn \tex_errorcontextlines:D { -1 }
21884     \tex_showbox:D \use:n {#2}
21885   \group_end:
21886   }
21887   {
21888     \__kernel_msg_error:nxx { kernel } { variable-not-defined }
21889     { \token_to_str:N #2 }
21890   }
21891 }
21892 \cs_generate_variant:Nn \__box_show:Nnn { NNff }

```

(End definition for `__box_show:Nnn`.)

38.10 Horizontal mode boxes

`\hbox:n` (The test suite for this command, and others in this file, is `m3box002.lvt`.)
Put a horizontal box directly into the input stream.

```

21893 \cs_new_protected:Npn \hbox:n #1
21894 { \tex_hbox:D \scan_stop: { \color_group_begin: #1 \color_group_end: } }

```

(End definition for `\hbox:n`. This function is documented on page 219.)

```

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn
21895 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
21896 \cs_new_protected:Npn \hbox_set:Nn #1#2
21897 {
21898   \tex_setbox:D #1 \tex_hbox:D
21899   { \color_group_begin: #2 \color_group_end: }
21900 }
21901 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
21902 \cs_new_protected:Npn \hbox_gset:Nn #1#2
21903 {
21904   \tex_global:D \tex_setbox:D #1 \tex_hbox:D
21905   { \color_group_begin: #2 \color_group_end: }
21906 }
21907 \cs_generate_variant:Nn \hbox_set:Nn { c }
21908 \cs_generate_variant:Nn \hbox_gset:Nn { c }

```

(End definition for `\hbox_set:Nn` and `\hbox_gset:Nn`. These functions are documented on page 219.)

`\hbox_set_to_wd:Nnn` Storing material in a horizontal box with a specified width. Again, put the dimension expression in parentheses when debugging.

```

\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn
21909 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
21910 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
21911 {
21912   \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
21913   { \color_group_begin: #3 \color_group_end: }
21914 }
21915 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
21916 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn #1#2#3
21917 {
21918   \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
21919   { \color_group_begin: #3 \color_group_end: }
21920 }
21921 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
21922 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }

```

(End definition for `\hbox_set_to_wd:Nnn` and `\hbox_gset_to_wd:Nnn`. These functions are documented on page 219.)

`\hbox_set:Nw` Storing material in a horizontal box. This type is useful in environment definitions.

```

\hbox_set:cw
\hbox_gset:Nw
\hbox_gset:cw
\hbox_set_end:
\hbox_gset_end:
21923 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
21924 \cs_new_protected:Npn \hbox_set:Nw #1
21925 {
21926   \tex_setbox:D #1 \tex_hbox:D
21927   \c_group_begin_token
21928   \color_group_begin:
21929 }
21930 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
21931 \cs_new_protected:Npn \hbox_gset:Nw #1
21932 {
21933   \tex_global:D \tex_setbox:D #1 \tex_hbox:D
21934   \c_group_begin_token
21935   \color_group_begin:
21936 }
21937 \cs_generate_variant:Nn \hbox_set:Nw { c }
21938 \cs_generate_variant:Nn \hbox_gset:Nw { c }
21939 \cs_new_protected:Npn \hbox_set_end:
21940 {
21941   \color_group_end:
21942   \c_group_end_token
21943 }
21944 \cs_new_eq:NN \hbox_gset_end: \hbox_set_end:

```

(End definition for `\hbox_set:Nw` and others. These functions are documented on page 220.)

`\hbox_set_to_wd:Nnw` Combining the above ideas.

```

\hbox_set_to_wd:cw
\hbox_gset_to_wd:Nnw
\hbox_gset_to_wd:cw
21945 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
21946 \cs_new_protected:Npn \hbox_set_to_wd:Nnw #1#2
21947 {
21948   \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
21949   \c_group_begin_token
21950   \color_group_begin:
21951 }

```

```

21952 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
21953 \cs_new_protected:Npn \hbox_gset_to_wd:Nnw #1#2
21954 {
21955   \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
21956   \c_group_begin_token
21957   \color_group_begin:
21958 }
21959 \cs_generate_variant:Nn \hbox_set_to_wd:Nnw { c }
21960 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnw { c }

```

(End definition for `\hbox_set_to_wd:Nnw` and `\hbox_gset_to_wd:Nnw`. These functions are documented on page 220.)

`\hbox_to_wd:nn` Put a horizontal box directly into the input stream.

```

\hbox_to_zero:n
21961 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
21962 {
21963   \tex_hbox:D to \__box_dim_eval:n {#1}
21964   { \color_group_begin: #2 \color_group_end: }
21965 }
21966 \cs_new_protected:Npn \hbox_to_zero:n #1
21967 {
21968   \tex_hbox:D to \c_zero_dim
21969   { \color_group_begin: #1 \color_group_end: }
21970 }

```

(End definition for `\hbox_to_wd:nn` and `\hbox_to_zero:n`. These functions are documented on page 219.)

`\hbox_overlap_left:n` Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

```

\hbox_overlap_right:n
21971 \cs_new_protected:Npn \hbox_overlap_left:n #1
21972 { \hbox_to_zero:n { \tex_hss:D #1 } }
21973 \cs_new_protected:Npn \hbox_overlap_right:n #1
21974 { \hbox_to_zero:n { #1 \tex_hss:D } }

```

(End definition for `\hbox_overlap_left:n` and `\hbox_overlap_right:n`. These functions are documented on page 220.)

`\hbox_unpack:N` Unpacking a box and if requested also clear it.

```

\hbox_unpack:c
\hbox_unpack_clear:N
\hbox_unpack_clear:c
21975 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
21976 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
21977 \cs_generate_variant:Nn \hbox_unpack:N { c }
21978 \cs_generate_variant:Nn \hbox_unpack_clear:N { c }

```

(End definition for `\hbox_unpack:N` and `\hbox_unpack_clear:N`. These functions are documented on page 220.)

38.11 Vertical mode boxes

TeX ends these boxes directly with the internal `end_graf` routine. This means that there is no `\par` at the end of vertical boxes unless we insert one.

`\vbox:n` The following test files are used for this code: `m3box003.lvt`.

The following test files are used for this code: *m3box003.lvt*.

\vbox_top:n Put a vertical box directly into the input stream.

```
21979 \cs_new_protected:Npn \vbox:n #1
21980 { \tex_vbox:D { \color_group_begin: #1 \color_group_end: } }
21981 \cs_new_protected:Npn \vbox_top:n #1
21982 { \tex_vtop:D { \color_group_begin: #1 \color_group_end: } }
```

(End definition for `\vbox:n` and `\vbox_top:n`. These functions are documented on page 220.)

\vbox_to_ht:nn Put a vertical box directly into the input stream.

```
\vbox_to_zero:n 21983 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
\vbox_to_ht:nn 21984 {
\vbox_to_zero:n 21985 \tex_vbox:D to \__box_dim_eval:n {#1}
21986 { \color_group_begin: #2 \color_group_end: }
21987 }
21988 \cs_new_protected:Npn \vbox_to_zero:n #1
21989 {
21990 \tex_vbox:D to \c_zero_dim
21991 { \color_group_begin: #1 \color_group_end: }
21992 }
```

(End definition for `\vbox_to_ht:nn` and others. These functions are documented on page 221.)

\vbox_set:Nn Storing material in a vertical box with a natural height.

```
\vbox_set:cn 21993 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\vbox_gset:Nn 21994 \cs_new_protected:Npn \vbox_set:Nn #1#2
\vbox_gset:cn 21995 {
21996 \tex_setbox:D #1 \tex_vbox:D
21997 { \color_group_begin: #2 \color_group_end: }
21998 }
21999 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
22000 \cs_new_protected:Npn \vbox_gset:Nn #1#2
22001 {
22002 \tex_global:D \tex_setbox:D #1 \tex_vbox:D
22003 { \color_group_begin: #2 \color_group_end: }
22004 }
22005 \cs_generate_variant:Nn \vbox_set:Nn { c }
22006 \cs_generate_variant:Nn \vbox_gset:Nn { c }
```

(End definition for `\vbox_set:Nn` and `\vbox_gset:Nn`. These functions are documented on page 221.)

\vbox_set_top:Nn Storing material in a vertical box with a natural height and reference point at the baseline of the first object in the box.

```
\vbox_set_top:cn 22007 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\vbox_gset_top:Nn 22008 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
\vbox_gset_top:cn 22009 {
22010 \tex_setbox:D #1 \tex_vtop:D
22011 { \color_group_begin: #2 \color_group_end: }
22012 }
22013 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
22014 \cs_new_protected:Npn \vbox_gset_top:Nn #1#2
22015 {
22016 \tex_global:D \tex_setbox:D #1 \tex_vtop:D
22017 { \color_group_begin: #2 \color_group_end: }
```

```

22018 }
22019 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
22020 \cs_generate_variant:Nn \vbox_gset_top:Nn { c }

```

(End definition for `\vbox_set_top:Nn` and `\vbox_gset_top:Nn`. These functions are documented on page 221.)

```

\vbox_set_to_ht:Nnn Storing material in a vertical box with a specified height.
\vbox_set_to_ht:cnn 22021 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\vbox_gset_to_ht:Nnn 22022 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
\vbox_gset_to_ht:cnn 22023 {
22024     \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
22025     { \color_group_begin: #3 \color_group_end: }
22026 }
22027 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
22028 \cs_new_protected:Npn \vbox_gset_to_ht:Nnn #1#2#3
22029 {
22030     \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
22031     { \color_group_begin: #3 \color_group_end: }
22032 }
22033 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }
22034 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }

```

(End definition for `\vbox_set_to_ht:Nnn` and `\vbox_gset_to_ht:Nnn`. These functions are documented on page 221.)

```

\vbox_set:Nw Storing material in a vertical box. This type is useful in environment definitions.
\vbox_set:cw 22035 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\vbox_gset:Nw 22036 \cs_new_protected:Npn \vbox_set:Nw #1
\vbox_gset:cw 22037 {
22038     \tex_setbox:D #1 \tex_vbox:D
22039     \c_group_begin_token
22040     \color_group_begin:
22041 }
22042 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
22043 \cs_new_protected:Npn \vbox_gset:Nw #1
22044 {
22045     \tex_global:D \tex_setbox:D #1 \tex_vbox:D
22046     \c_group_begin_token
22047     \color_group_begin:
22048 }
22049 \cs_generate_variant:Nn \vbox_set:Nw { c }
22050 \cs_generate_variant:Nn \vbox_gset:Nw { c }
22051 \cs_new_protected:Npn \vbox_set_end:
22052 {
22053     \color_group_end:
22054     \c_group_end_token
22055 }
22056 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:

```

(End definition for `\vbox_set:Nw` and others. These functions are documented on page 221.)

```

\vbox_set_to_ht:Nnw A combination of the above ideas.
\vbox_set_to_ht:cnw 22057 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\vbox_gset_to_ht:Nnw 22058 \cs_new_protected:Npn \vbox_set_to_ht:Nnw #1#2
\vbox_gset_to_ht:cnw

```

```

22059 {
22060   \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
22061   \c_group_begin_token
22062   \color_group_begin:
22063 }
22064 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
22065 \cs_new_protected:Npn \vbox_gset_to_ht:Nnw #1#2
22066 {
22067   \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \_box_dim_eval:n {#2}
22068   \c_group_begin_token
22069   \color_group_begin:
22070 }
22071 \cs_generate_variant:Nn \vbox_set_to_ht:Nnw { c }
22072 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnw { c }

```

(End definition for `\vbox_set_to_ht:Nnw` and `\vbox_gset_to_ht:Nnw`. These functions are documented on page 221.)

```

\ vbox_unpack:N      Unpacking a box and if requested also clear it.
\ vbox_unpack:c      22073 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\ vbox_unpack_clear:N 22074 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
\ vbox_unpack_clear:c 22075 \cs_generate_variant:Nn \vbox_unpack:N { c }
                     22076 \cs_generate_variant:Nn \vbox_unpack_clear:N { c }

```

(End definition for `\vbox_unpack:N` and `\vbox_unpack_clear:N`. These functions are documented on page 222.)

```

\ vbox_set_split_to_ht:NNn Splitting a vertical box in two.
22077 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
22078 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
22079 { \tex_setbox:D #1 \tex_vsplit:D #2 to \_box_dim_eval:n {#3} }

```

(End definition for `\vbox_set_split_to_ht:NNn`. This function is documented on page 221.)

38.12 Affine transformations

`\l__box_angle_fp` When rotating boxes, the angle itself may be needed by the engine-dependent code. This is done using the `fp` module so that the value is tidied up properly.

```
22080 \fp_new:N \l__box_angle_fp
```

(End definition for `\l__box_angle_fp`.)

`\l__box_cos_fp` `\l__box_sin_fp` These are used to hold the calculated sine and cosine values while carrying out a rotation.

```

22081 \fp_new:N \l__box_cos_fp
22082 \fp_new:N \l__box_sin_fp

```

(End definition for `\l__box_cos_fp` and `\l__box_sin_fp`.)

```

\l__box_top_dim      These are the positions of the four edges of a box before manipulation.
\l__box_bottom_dim  22083 \dim_new:N \l__box_top_dim
\l__box_left_dim     22084 \dim_new:N \l__box_bottom_dim
\l__box_right_dim    22085 \dim_new:N \l__box_left_dim
                     22086 \dim_new:N \l__box_right_dim

```

(End definition for `\l__box_top_dim` and others.)

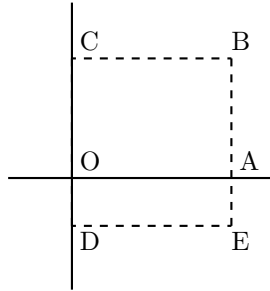


Figure 1: Co-ordinates of a box prior to rotation.

```

\l__box_top_new_dim  These are the positions of the four edges of a box after manipulation.
\l__box_bottom_new_dim 22087 \dim_new:N \l__box_top_new_dim
\l__box_left_new_dim 22088 \dim_new:N \l__box_bottom_new_dim
\l__box_right_new_dim 22089 \dim_new:N \l__box_left_new_dim
22090 \dim_new:N \l__box_right_new_dim

(End definition for \l__box_top_new_dim and others.)

\l__box_internal_box Scratch space, but also needed by some parts of the driver.
22091 \box_new:N \l__box_internal_box

(End definition for \l__box_internal_box.)

\box_rotate:Nn Rotation of a box starts with working out the relevant sine and cosine. The actual
  rotation is in an auxiliary to keep the flow slightly clearer
  \__box_rotate:N
    \__box_rotate_x:nnN 22092 \cs_new_protected:Npn \box_rotate:Nn #1#2
    \__box_rotate_y:nnN 22093 {
      \__box_rotate_quadrant_one: 22094 \hbox_set:Nn #1
      \__box_rotate_quadrant_two: 22095 {
        \__box_rotate_quadrant_three: 22096 \fp_set:Nn \l__box_angle_fp {#2}
        \__box_rotate_quadrant_four: 22097 \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
        22098 \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
        22099 \__box_rotate:N #1
        22100 }
        22101 }
    }

The edges of the box are then recorded: the left edge is always at zero. Rotation of the
four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.
22102 \cs_new_protected:Npn \__box_rotate:N #1
22103 {
22104 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
22105 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
22106 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
22107 \dim_zero:N \l__box_left_dim

```

The next step is to work out the x and y coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices B , C , D and E is illustrated (Figure 1). The vertex O is the reference point on the baseline, and in this implementation is also the centre of rotation. The formulae are, for a point P and

angle α :

$$\begin{aligned}
 P'_x &= P_x - O_x \\
 P'_y &= P_y - O_y \\
 P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\
 P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\
 P'''_x &= P''_x + O_x + L_x \\
 P'''_y &= P''_y + O_y
 \end{aligned}$$

The “extra” horizontal translation L_x at the end is calculated so that the leftmost point of the resulting box has x -coordinate 0. This is desirable as T_EX boxes must have the reference point at the left edge of the box. (As O is always $(0,0)$, this part of the calculation is omitted here.)

```

22108 \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
22109 {
22110     \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
22111     { \__box_rotate_quadrant_one: }
22112     { \__box_rotate_quadrant_two: }
22113 }
22114 {
22115     \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
22116     { \__box_rotate_quadrant_three: }
22117     { \__box_rotate_quadrant_four: }
22118 }

```

The position of the box edges are now known, but the box at this stage be misplaced relative to the current T_EX reference point. So the content of the box is moved such that the reference point of the rotated box is in the same place as the original.

```

22119 \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
22120 \hbox_set:Nn \l__box_internal_box
22121 {
22122     \tex_kern:D -\l__box_left_new_dim
22123     \hbox:n
22124     {
22125         \driver_box_use_rotate:Nn
22126         \l__box_internal_box
22127         \l__box_angle_fp
22128     }
22129 }

```

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

```

22130 \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
22131 \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
22132 \box_set_wd:Nn \l__box_internal_box
22133 { \l__box_right_new_dim - \l__box_left_new_dim }
22134 \box_use_drop:N \l__box_internal_box
22135 }

```

These functions take a general point ($\#1, \#2$) and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both x' and y' at the same time. Contrast this with the equivalent function in the `l3coffins` module, where both parts are needed.

```

22136 \cs_new_protected:Npn \__box_rotate_x:nnN #1#2#3
22137 {
22138   \dim_set:Nn #3
22139   {
22140     \fp_to_dim:n
22141     {
22142       \l__box_cos_fp * \dim_to_fp:n {#1}
22143       - \l__box_sin_fp * \dim_to_fp:n {#2}
22144     }
22145   }
22146 }
22147 \cs_new_protected:Npn \__box_rotate_y:nnN #1#2#3
22148 {
22149   \dim_set:Nn #3
22150   {
22151     \fp_to_dim:n
22152     {
22153       \l__box_sin_fp * \dim_to_fp:n {#1}
22154       + \l__box_cos_fp * \dim_to_fp:n {#2}
22155     }
22156   }
22157 }

```

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting y -values, whereas the left and right edges need the x -values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

```

22158 \cs_new_protected:Npn \__box_rotate_quadrant_one:
22159 {
22160   \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
22161   \l__box_top_new_dim
22162   \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
22163   \l__box_bottom_new_dim
22164   \__box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
22165   \l__box_left_new_dim
22166   \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
22167   \l__box_right_new_dim
22168 }
22169 \cs_new_protected:Npn \__box_rotate_quadrant_two:
22170 {
22171   \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
22172   \l__box_top_new_dim
22173   \__box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
22174   \l__box_bottom_new_dim
22175   \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
22176   \l__box_left_new_dim
22177   \__box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
22178   \l__box_right_new_dim
22179 }
22180 \cs_new_protected:Npn \__box_rotate_quadrant_three:
22181 {
22182   \__box_rotate_y:nnN \l__box_left_dim \l__box_bottom_dim
22183   \l__box_top_new_dim

```

```

22184     \_box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
22185     \l__box_bottom_new_dim
22186     \_box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
22187     \l__box_left_new_dim
22188     \_box_rotate_x:nnN \l__box_left_dim \l__box_top_dim
22189     \l__box_right_new_dim
22190   }
22191 \cs_new_protected:Npn \_box_rotate_quadrant_four:
22192 {
22193     \_box_rotate_y:nnN \l__box_left_dim \l__box_top_dim
22194     \l__box_top_new_dim
22195     \_box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
22196     \l__box_bottom_new_dim
22197     \_box_rotate_x:nnN \l__box_left_dim \l__box_bottom_dim
22198     \l__box_left_new_dim
22199     \_box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
22200     \l__box_right_new_dim
22201 }

```

(End definition for `\box_rotate:Nn` and others. This function is documented on page 224.)

`\l__box_scale_x_fp` Scaling is potentially-different in the two axes.

```

\l__box_scale_y_fp 22202 \fp_new:N \l__box_scale_x_fp
22203 \fp_new:N \l__box_scale_y_fp

```

(End definition for `\l__box_scale_x_fp` and `\l__box_scale_y_fp`.)

`\box_resize_to_wd_and_ht_plus_dp:Nnn`

Resizing a box starts by working out the various dimensions of the existing box.

```

\box_resize_to_wd_and_ht_plus_dp:cmn 22204 \cs_new_protected:Npn \box_resize_to_wd_and_ht_plus_dp:Nnn #1#2#3
\__box_resize_set_corners:N 22205 {
\__box_resize:N 22206     \hbox_set:Nn #1
\__box_resize:NNN 22207     {
22208         \__box_resize_set_corners:N #1

```

The x -scaling and resulting box size is easy enough to work out: the dimension is that given as #2, and the scale is simply the new width divided by the old one.

```

22209         \fp_set:Nn \l__box_scale_x_fp
22210         { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }

```

The y -scaling needs both the height and the depth of the current box.

```

22211         \fp_set:Nn \l__box_scale_y_fp
22212         {
22213             \dim_to_fp:n {#3}
22214             / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
22215         }

```

Hand off to the auxiliary which does the rest of the work.

```

22216         \__box_resize:N #1
22217     }
22218 }
22219 \cs_generate_variant:Nn \box_resize_to_wd_and_ht_plus_dp:Nnn { c }
22220 \cs_new_protected:Npn \__box_resize_set_corners:N #1
22221 {
22222     \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
22223     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
22224     \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }

```

```

22225 \dim_zero:N \l__box_left_dim
22226 }

```

With at least one real scaling to do, the next phase is to find the new edge co-ordinates. In the x direction this is relatively easy: just scale the right edge. In the y direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```

22227 \cs_new_protected:Npn \__box_resize:N #1
22228 {
22229   \__box_resize:NNN \l__box_right_new_dim
22230   \l__box_scale_x_fp \l__box_right_dim
22231   \__box_resize:NNN \l__box_bottom_new_dim
22232   \l__box_scale_y_fp \l__box_bottom_dim
22233   \__box_resize:NNN \l__box_top_new_dim
22234   \l__box_scale_y_fp \l__box_top_dim
22235   \__box_resize_common:N #1
22236 }
22237 \cs_new_protected:Npn \__box_resize:NNN #1#2#3
22238 {
22239   \dim_set:Nn #1
22240     { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
22241 }

```

(End definition for `\box_resize_to_wd_and_ht_plus_dp:Nnn` and others. This function is documented on page 224.)

<pre> \box_resize_to_ht:Nn \box_resize_to_ht:cn \box_resize_to_ht_plus_dp:Nn \box_resize_to_ht_plus_dp:cn \box_resize_to_wd:Nn \box_resize_to_wd:cn \box_resize_to_wd_and_ht:Nnn \box_resize_to_wd_and_ht:cnn </pre>	<p>Scaling to a (total) height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case).</p> <pre> 22242 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2 22243 { 22244 \hbox_set:Nn #1 22245 { 22246 __box_resize_set_corners:N #1 22247 \fp_set:Nn \l__box_scale_y_fp 22248 { 22249 \dim_to_fp:n {#2} 22250 / \dim_to_fp:n { \l__box_top_dim } 22251 } 22252 \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp 22253 __box_resize:N #1 22254 } 22255 } 22256 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c } 22257 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2 22258 { 22259 \hbox_set:Nn #1 22260 { 22261 __box_resize_set_corners:N #1 22262 \fp_set:Nn \l__box_scale_y_fp 22263 { 22264 \dim_to_fp:n {#2} 22265 / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim } </pre>
--	---

```

22266     }
22267     \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
22268     \__box_resize:N #1
22269   }
22270 }
22271 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
22272 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
22273 {
22274   \hbox_set:Nn #1
22275   {
22276     \__box_resize_set_corners:N #1
22277     \fp_set:Nn \l__box_scale_x_fp
22278     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
22279     \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
22280     \__box_resize:N #1
22281   }
22282 }
22283 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
22284 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
22285 {
22286   \hbox_set:Nn #1
22287   {
22288     \__box_resize_set_corners:N #1
22289     \fp_set:Nn \l__box_scale_x_fp
22290     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
22291     \fp_set:Nn \l__box_scale_y_fp
22292     {
22293       \dim_to_fp:n {#3}
22294       / \dim_to_fp:n { \l__box_top_dim }
22295     }
22296     \__box_resize:N #1
22297   }
22298 }
22299 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }

```

(End definition for `\box_resize_to_ht:Nn` and others. These functions are documented on page 223.)

`\box_scale:Nnn` When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases.

`\box_scale:cnn` Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. The code here is split into two as this allows sharing with the auto-resizing functions.

`__box_scale_aux:N`

```

22300 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
22301 {
22302   \hbox_set:Nn #1
22303   {
22304     \fp_set:Nn \l__box_scale_x_fp {#2}
22305     \fp_set:Nn \l__box_scale_y_fp {#3}
22306     \__box_scale_aux:N #1
22307   }
22308 }
22309 \cs_generate_variant:Nn \box_scale:Nnn { c }
22310 \cs_new_protected:Npn \__box_scale_aux:N #1
22311 {

```

```

22312 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
22313 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
22314 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
22315 \dim_zero:N \l__box_left_dim
22316 \dim_set:Nn \l__box_top_new_dim
22317 { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
22318 \dim_set:Nn \l__box_bottom_new_dim
22319 { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
22320 \dim_set:Nn \l__box_right_new_dim
22321 { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
22322 \__box_resize_common:N #1
22323 }

```

(End definition for `\box_scale:Nnn` and `__box_scale_aux:N`. This function is documented on page 224.)

`\box_autosize_to_wd_and_ht:Nnn` Although autosizing a box uses dimensions, it has more in common in implementation with scaling. As such, most of the real work here is done elsewhere.

```

\box_autosize_to_wd_and_ht:cmn
\box_autosize_to_wd_and_ht_plus_dp:Nnn
\box_autosize_to_wd_and_ht_plus_dp:cmn
\__box_autosize:Nnnn
22324 \cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
22325 { \__box_autosize:Nnnn #1 {#2} {#3} { \box_ht:N #1 } }
22326 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
22327 \cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
22328 { \__box_autosize:Nnnn #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 } }
22329 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht_plus_dp:Nnn { c }
22330 \cs_new_protected:Npn \__box_autosize:Nnnn #1#2#3#4
22331 {
22332 \hbox_set:Nn #1
22333 {
22334 \fp_set:Nn \l__box_scale_x_fp { ( #2 ) / \box_wd:N #1 }
22335 \fp_set:Nn \l__box_scale_y_fp { ( #3 ) / ( #4 ) }
22336 \fp_compare:nNnTF \l__box_scale_x_fp > \l__box_scale_y_fp
22337 { \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp }
22338 { \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp }
22339 \__box_scale_aux:N #1
22340 }
22341 }

```

(End definition for `\box_autosize_to_wd_and_ht:Nnn`, `\box_autosize_to_wd_and_ht_plus_dp:Nnn`, and `__box_autosize:Nnnn`. These functions are documented on page 222.)

`__box_resize_common:N` The main resize function places its input into a box which start off with zero width, and includes the handles for engine rescaling.

```

22342 \cs_new_protected:Npn \__box_resize_common:N #1
22343 {
22344 \hbox_set:Nn \l__box_internal_box
22345 {
22346 \driver_box_use_scale:Nnn
22347 #1
22348 \l__box_scale_x_fp
22349 \l__box_scale_y_fp
22350 }

```

The new height and depth can be applied directly.

```

22351 \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
22352 {

```

```

22353     \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
22354     \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
22355   }
22356   {
22357     \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
22358     \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
22359   }

```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```

22360   \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
22361   {
22362     \hbox_to_wd:nn { \l__box_right_new_dim }
22363     {
22364       \tex_kern:D \l__box_right_new_dim
22365       \box_use_drop:N \l__box_internal_box
22366       \tex_hss:D
22367     }
22368   }
22369   {
22370     \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
22371     \hbox:n
22372     {
22373       \tex_kern:D \c_zero_dim
22374       \box_use_drop:N \l__box_internal_box
22375       \tex_hss:D
22376     }
22377   }
22378 }

```

(End definition for `__box_resize_common:N`.)

38.13 Deprecated functions

```

\box_resize:Nnn
\box_resize:cnn
\box_use_clear:N
\box_use_clear:c
22379 \__kernel_patch_deprecation:nnNNpn
22380 { 2018-12-31 } { \box_resize_to_wd_and_ht_plus_dp:Nnn }
22381 \cs_new_protected:Npn \box_resize:Nnn
22382 { \box_resize_to_wd_and_ht_plus_dp:Nnn }
22383 \__kernel_patch_deprecation:nnNNpn
22384 { 2018-12-31 } { \box_resize_to_wd_and_ht_plus_dp:cnn }
22385 \cs_new_protected:Npn \box_resize:cnn
22386 { \box_resize_to_wd_and_ht_plus_dp:cnn }
22387 \__kernel_patch_deprecation:nnNNpn
22388 { 2018-12-31 } { \box_use_drop:N }
22389 \cs_new_protected:Npn \box_use_clear:N { \box_use_drop:N }
22390 \__kernel_patch_deprecation:nnNNpn
22391 { 2018-12-31 } { \box_use_drop:c }
22392 \cs_new_protected:Npn \box_use_clear:c { \box_use_drop:c }

```

(End definition for `\box_resize:Nnn` and `\box_use_clear:N`.)

```

22393 </initex | package>

```

39 l3coffins Implementation

22394 $\langle *initex | package \rangle$

22395 $\langle @@=coffin \rangle$

39.1 Coffins: data structures and general variables

$\backslash l_coffin_internal_box$ Scratch variables.

$\backslash l_coffin_internal_dim$ 22396 $\backslash box_new:N \backslash l_coffin_internal_box$

$\backslash l_coffin_internal_tl$ 22397 $\backslash dim_new:N \backslash l_coffin_internal_dim$

22398 $\backslash tl_new:N \backslash l_coffin_internal_tl$

(End definition for $\backslash l_coffin_internal_box$, $\backslash l_coffin_internal_dim$, and $\backslash l_coffin_internal_tl$.)

$\backslash c_coffin_corners_prop$ The “corners”; of a coffin define the real content, as opposed to the TeX bounding box. They all start off in the same place, of course.

22399 $\backslash prop_const_from_keyval:Nn \backslash c_coffin_corners_prop$

22400 {
22401 $tl = \{ Opt \} \{ Opt \} ,$
22402 $tr = \{ Opt \} \{ Opt \} ,$
22403 $bl = \{ Opt \} \{ Opt \} ,$
22404 $br = \{ Opt \} \{ Opt \} ,$
22405 }

(End definition for $\backslash c_coffin_corners_prop$.)

$\backslash c_coffin_poles_prop$ Pole positions are given for horizontal, vertical and reference-point based values.

22406 $\backslash prop_const_from_keyval:Nn \backslash c_coffin_poles_prop$

22407 {
22408 $l = \{ Opt \} \{ Opt \} \{ Opt \} \{ 1000pt \} ,$
22409 $hc = \{ Opt \} \{ Opt \} \{ Opt \} \{ 1000pt \} ,$
22410 $r = \{ Opt \} \{ Opt \} \{ Opt \} \{ 1000pt \} ,$
22411 $b = \{ Opt \} \{ Opt \} \{ 1000pt \} \{ Opt \} ,$
22412 $vc = \{ Opt \} \{ Opt \} \{ 1000pt \} \{ Opt \} ,$
22413 $t = \{ Opt \} \{ Opt \} \{ 1000pt \} \{ Opt \} ,$
22414 $B = \{ Opt \} \{ Opt \} \{ 1000pt \} \{ Opt \} ,$
22415 $H = \{ Opt \} \{ Opt \} \{ 1000pt \} \{ Opt \} ,$
22416 $T = \{ Opt \} \{ Opt \} \{ 1000pt \} \{ Opt \} ,$
22417 }

(End definition for $\backslash c_coffin_poles_prop$.)

$\backslash l_coffin_slope_x_fp$ Used for calculations of intersections.

$\backslash l_coffin_slope_y_fp$ 22418 $\backslash fp_new:N \backslash l_coffin_slope_x_fp$

22419 $\backslash fp_new:N \backslash l_coffin_slope_y_fp$

(End definition for $\backslash l_coffin_slope_x_fp$ and $\backslash l_coffin_slope_y_fp$.)

$\backslash l_coffin_error_bool$ For propagating errors so that parts of the code can work around them.

22420 $\backslash bool_new:N \backslash l_coffin_error_bool$

(End definition for $\backslash l_coffin_error_bool$.)

`\l__coffin_offset_x_dim` The offset between two sets of coffin handles when typesetting. These values are corrected from those requested in an alignment for the positions of the handles.

22421 `\dim_new:N \l__coffin_offset_x_dim`

22422 `\dim_new:N \l__coffin_offset_y_dim`

(End definition for `\l__coffin_offset_x_dim` and `\l__coffin_offset_y_dim`.)

`\l__coffin_pole_a_tl` Needed for finding the intersection of two poles.

`\l__coffin_pole_b_tl` 22423 `\tl_new:N \l__coffin_pole_a_tl`

22424 `\tl_new:N \l__coffin_pole_b_tl`

(End definition for `\l__coffin_pole_a_tl` and `\l__coffin_pole_b_tl`.)

`\l__coffin_x_dim` For calculating intersections and so forth.

`\l__coffin_y_dim` 22425 `\dim_new:N \l__coffin_x_dim`

`\l__coffin_x_prime_dim` 22426 `\dim_new:N \l__coffin_y_dim`

`\l__coffin_y_prime_dim` 22427 `\dim_new:N \l__coffin_x_prime_dim`

22428 `\dim_new:N \l__coffin_y_prime_dim`

(End definition for `\l__coffin_x_dim` and others.)

39.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

`__coffin_to_value:N` Coffins are a two-part structure and we rely on the internal nature of box allocation to make everything work. As such, we need an interface to turn coffin identifiers into numbers. For the purposes here, the signature allowed is N despite the nature of the underlying primitive.

22429 `\cs_new_eq:NN __coffin_to_value:N \tex_number:D`

(End definition for `__coffin_to_value:N`.)

`\coffin_if_exist_p:N` Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

`\coffin_if_exist:N`

`\coffin_if_exist:N`

`\coffin_if_exist:cTF`

22430 `\prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }`

22431 `{`

22432 `\cs_if_exist:NTF #1`

22433 `{`

22434 `\cs_if_exist:cTF { l__coffin_poles_ __coffin_to_value:N #1 _prop }`

22435 `{ \prg_return_true: }`

22436 `{ \prg_return_false: }`

22437 `}`

22438 `{ \prg_return_false: }`

22439 `}`

22440 `\prg_generate_conditional_variant:Nnn \coffin_if_exist:N`

22441 `{ c } { p , T , F , TF }`

(End definition for `\coffin_if_exist:N`. This function is documented on page 226.)

`__coffin_if_exist:NT` Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. So a wrapper is provided to deal with this correctly, issuing an error on erroneous use.

```

22442 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
22443 {
22444   \coffin_if_exist:NTF #1
22445     { #2 }
22446     {
22447       \__kernel_msg_error:nnx { kernel } { unknown-coffin }
22448       { \token_to_str:N #1 }
22449     }
22450 }
```

(End definition for `__coffin_if_exist:NT`.)

`\coffin_clear:N` Clearing coffins means emptying the box and resetting all of the structures.

```

\coffin_clear:c 22451 \cs_new_protected:Npn \coffin_clear:N #1
22452 {
22453   \__coffin_if_exist:NT #1
22454   {
22455     \box_clear:N #1
22456     \__coffin_reset_structure:N #1
22457   }
22458 }
22459 \cs_generate_variant:Nn \coffin_clear:N { c }
```

(End definition for `\coffin_clear:N`. This function is documented on page 226.)

`\coffin_new:N` Creating a new coffin means making the underlying box and adding the data structures.

`\coffin_new:c` These are created globally, as there is a need to avoid any strange effects if the coffin is created inside a group. This means that the usual rule about `\l_...` variables has to be broken. The `\debug_suspend:` and `\debug_resume:` functions prevent these checks. They also prevent `\prop_clear_new:c` from writing useless information to the log file.

```

22460 \cs_new_protected:Npn \coffin_new:N #1
22461 {
22462   \box_new:N #1
22463   \debug_suspend:
22464   \prop_clear_new:c { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
22465   \prop_clear_new:c { l__coffin_poles_ \__coffin_to_value:N #1 _prop }
22466   \prop_gset_eq:cN { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
22467     \c__coffin_corners_prop
22468   \prop_gset_eq:cN { l__coffin_poles_ \__coffin_to_value:N #1 _prop }
22469     \c__coffin_poles_prop
22470   \debug_resume:
22471 }
22472 \cs_generate_variant:Nn \coffin_new:N { c }
```

(End definition for `\coffin_new:N`. This function is documented on page 226.)

`\hcoffin_set:Nn` Horizontal coffins are relatively easy: set the appropriate box, reset the structures then update the handle positions.

```

\hcoffin_set:cn 22473 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
22474 {
22475   \__coffin_if_exist:NT #1
```

```

22476     {
22477         \hbox_set:Nn #1
22478         {
22479             \color_ensure_current:
22480             #2
22481         }
22482         \__coffin_reset_structure:N #1
22483         \__coffin_update_poles:N #1
22484         \__coffin_update_corners:N #1
22485     }
22486 }
22487 \cs_generate_variant:Nn \hcoffin_set:Nn { c }

```

(End definition for `\hcoffin_set:Nn`. This function is documented on page 226.)

`\vcoffin_set:Nnn` Setting vertical coffins is more complex. First, the material is typeset with a given width. **`\vcoffin_set:cnn`** The default handles and poles are set as for a horizontal coffin, before finding the top baseline using a temporary box. No `\color_ensure_current:` here as that would add a whatsit to the start of the vertical box and mess up the location of the T pole (see *TEX by Topic* for discussion of the `\vtop` primitive, used to do the measuring).

```

22488 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
22489 {
22490     \__coffin_if_exist:NT #1
22491     {
22492         \vbox_set:Nn #1
22493         {
22494             \dim_set:Nn \tex_hsize:D {#2}
22495             (*package)
22496             \dim_set_eq:NN \linewidth \tex_hsize:D
22497             \dim_set_eq:NN \columnwidth \tex_hsize:D
22498             /package)
22499             #3
22500         }
22501         \__coffin_reset_structure:N #1
22502         \__coffin_update_poles:N #1
22503         \__coffin_update_corners:N #1
22504         \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
22505         \__coffin_set_pole:Nnx #1 { T }
22506         {
22507             { 0pt }
22508             {
22509                 \dim_eval:n
22510                 { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
22511             }
22512             { 1000pt }
22513             { 0pt }
22514         }
22515         \box_clear:N \l__coffin_internal_box
22516     }
22517 }
22518 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }

```

(End definition for `\vcoffin_set:Nnn`. This function is documented on page 227.)

`\hcoffin_set:Nw` These are the “begin”/“end” versions of the above: watch the grouping!

```

\hcoffin_set:cnw 22519 \cs_new_protected:Npn \hcoffin_set:Nw #1
\hcoffin_set_end: 22520 {
22521   \__coffin_if_exist:NT #1
22522   {
22523     \hbox_set:Nw #1 \color_ensure_current:
22524     \cs_set_protected:Npn \hcoffin_set_end:
22525     {
22526       \hbox_set_end:
22527       \__coffin_reset_structure:N #1
22528       \__coffin_update_poles:N #1
22529       \__coffin_update_corners:N #1
22530     }
22531   }
22532 }
22533 \cs_new_protected:Npn \hcoffin_set_end: { }
22534 \cs_generate_variant:Nn \hcoffin_set:Nw { c }

```

(End definition for `\hcoffin_set:Nw` and `\hcoffin_set_end:`. These functions are documented on page 226.)

`\vcoffin_set:Nnw` The same for vertical coffins.

```

\vcoffin_set:cnw 22535 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
\vcoffin_set_end: 22536 {
22537   \__coffin_if_exist:NT #1
22538   {
22539     \vbox_set:Nw #1
22540     \dim_set:Nn \tex_hsize:D {#2}
22541     (*package)
22542     \dim_set_eq:NN \linewidth \tex_hsize:D
22543     \dim_set_eq:NN \columnwidth \tex_hsize:D
22544     (/package)
22545     \cs_set_protected:Npn \vcoffin_set_end:
22546     {
22547       \vbox_set_end:
22548       \__coffin_reset_structure:N #1
22549       \__coffin_update_poles:N #1
22550       \__coffin_update_corners:N #1
22551       \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
22552       \__coffin_set_pole:Nnx #1 { T }
22553       {
22554         { Opt }
22555         {
22556           \dim_eval:n
22557           { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
22558         }
22559         { 1000pt }
22560         { Opt }
22561       }
22562       \box_clear:N \l__coffin_internal_box
22563     }
22564   }
22565 }
22566 \cs_new_protected:Npn \vcoffin_set_end: { }
22567 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }

```

(End definition for `\vcoffin_set:Nnw` and `\vcoffin_set_end:.`. These functions are documented on page 227.)

`\coffin_set_eq:NN` Setting two coffins equal is just a wrapper around other functions.

```

\coffin_set_eq:Nc 22568 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
\coffin_set_eq:cN 22569 {
\coffin_set_eq:cc 22570   \__coffin_if_exist:NT #1
22571   {
22572     \box_set_eq:NN #1 #2
22573     \__coffin_set_eq_structure:NN #1 #2
22574   }
22575 }
22576 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
```

(End definition for `\coffin_set_eq:NN`. This function is documented on page 226.)

`\c_empty_coffin` Special coffins: these cannot be set up earlier as they need `\coffin_new:N`. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available. The empty coffin is created entirely by hand: not everything is in place yet.

```

\l__coffin_aligned_coffin 22577 \coffin_new:N \c_empty_coffin
\l__coffin_aligned_internal_coffin 22578 \tex_setbox:D \c_empty_coffin = \tex_hbox:D { }
22579 \coffin_new:N \l__coffin_aligned_coffin
22580 \coffin_new:N \l__coffin_aligned_internal_coffin
```

(End definition for `\c_empty_coffin`, `\l__coffin_aligned_coffin`, and `\l__coffin_aligned_internal_coffin`. This variable is documented on page 229.)

`\l_tmpa_coffin` The usual scratch space.

```

\l_tmpb_coffin 22581 \coffin_new:N \l_tmpa_coffin
22582 \coffin_new:N \l_tmpb_coffin
```

(End definition for `\l_tmpa_coffin` and `\l_tmpb_coffin`. These variables are documented on page 229.)

39.3 Measuring coffins

`\coffin_dp:N` Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

```

\coffin_dp:c 22583 \cs_new_eq:NN \coffin_dp:N \box_dp:N
\coffin_ht:c 22584 \cs_new_eq:NN \coffin_dp:c \box_dp:c
\coffin_ht:N 22585 \cs_new_eq:NN \coffin_ht:N \box_ht:N
\coffin_wd:N 22586 \cs_new_eq:NN \coffin_ht:c \box_ht:c
\coffin_wd:c 22587 \cs_new_eq:NN \coffin_wd:N \box_wd:N
22588 \cs_new_eq:NN \coffin_wd:c \box_wd:c
```

(End definition for `\coffin_dp:N`, `\coffin_ht:N`, and `\coffin_wd:N`. These functions are documented on page 228.)

39.4 Coffins: handle and pole management

`__coffin_get_pole:NnN` A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

```

22589 \cs_new_protected:Npn __coffin_get_pole:NnN #1#2#3
22590 {
22591   \prop_get:cnNF
22592     { l__coffin_poles_ __coffin_to_value:N #1 _prop } {#2} #3
22593   {
22594     \__kernel_msg_error:nxxx { kernel } { unknown-coffin-pole }
22595     {#2} { \token_to_str:N #1 }
22596     \tl_set:Nn #3 { { Opt } { Opt } { Opt } { Opt } }
22597   }
22598 }
```

(End definition for `__coffin_get_pole:NnN`.)

`__coffin_reset_structure:N` Resetting the structure is a simple copy job.

```

22599 \cs_new_protected:Npn __coffin_reset_structure:N #1
22600 {
22601   \prop_set_eq:cN { l__coffin_corners_ __coffin_to_value:N #1 _prop }
22602   \c__coffin_corners_prop
22603   \prop_set_eq:cN { l__coffin_poles_ __coffin_to_value:N #1 _prop }
22604   \c__coffin_poles_prop
22605 }
```

(End definition for `__coffin_reset_structure:N`.)

`__coffin_set_eq_structure:NN` Setting coffin structures equal simply means copying the property list.

```

__coffin_gset_eq_structure:NN
22606 \cs_new_protected:Npn __coffin_set_eq_structure:NN #1#2
22607 {
22608   \prop_set_eq:cc { l__coffin_corners_ __coffin_to_value:N #1 _prop }
22609   { l__coffin_corners_ __coffin_to_value:N #2 _prop }
22610   \prop_set_eq:cc { l__coffin_poles_ __coffin_to_value:N #1 _prop }
22611   { l__coffin_poles_ __coffin_to_value:N #2 _prop }
22612 }
22613 \cs_new_protected:Npn __coffin_gset_eq_structure:NN #1#2
22614 {
22615   \prop_gset_eq:cc { l__coffin_corners_ __coffin_to_value:N #1 _prop }
22616   { l__coffin_corners_ __coffin_to_value:N #2 _prop }
22617   \prop_gset_eq:cc { l__coffin_poles_ __coffin_to_value:N #1 _prop }
22618   { l__coffin_poles_ __coffin_to_value:N #2 _prop }
22619 }
```

(End definition for `__coffin_set_eq_structure:NN` and `__coffin_gset_eq_structure:NN`.)

`\coffin_set_horizontal_pole:Nnn` Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

```

\coffin_set_horizontal_pole:cnm
\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnm
__coffin_set_pole:Nnn
__coffin_set_pole:Nnx
22620 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
22621 {
22622   __coffin_if_exist:NT #1
22623   {
22624     __coffin_set_pole:Nnx #1 {#2}
```

```

22625         {
22626         { Opt } { \dim_eval:n {#3} }
22627         { 1000pt } { Opt }
22628         }
22629     }
22630 }
22631 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
22632 {
22633     \__coffin_if_exist:NT #1
22634     {
22635         \__coffin_set_pole:Nnx #1 {#2}
22636         {
22637             { \dim_eval:n {#3} } { Opt }
22638             { Opt } { 1000pt }
22639         }
22640     }
22641 }
22642 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
22643 {
22644     \prop_put:cnn { l__coffin_poles_ \__coffin_to_value:N #1 _prop }
22645     {#2} {#3}
22646 }
22647 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
22648 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
22649 \cs_generate_variant:Nn \__coffin_set_pole:Nnn { Nnx }

```

(End definition for `\coffin_set_horizontal_pole:Nnn`, `\coffin_set_vertical_pole:Nnn`, and `__coffin_set_pole:Nnn`. These functions are documented on page 227.)

`__coffin_update_corners:N` Updating the corners of a coffin is straight-forward as at this stage there can be no rotation. So the corners of the content are just those of the underlying \TeX box.

```

22650 \cs_new_protected:Npn \__coffin_update_corners:N #1
22651 {
22652     \prop_put:cnx { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
22653     { tl }
22654     { { Opt } { \dim_eval:n { \box_ht:N #1 } } }
22655     \prop_put:cnx { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
22656     { tr }
22657     {
22658         { \dim_eval:n { \box_wd:N #1 } }
22659         { \dim_eval:n { \box_ht:N #1 } }
22660     }
22661     \prop_put:cnx { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
22662     { bl }
22663     { { Opt } { \dim_eval:n { -\box_dp:N #1 } } }
22664     \prop_put:cnx { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
22665     { br }
22666     {
22667         { \dim_eval:n { \box_wd:N #1 } }
22668         { \dim_eval:n { -\box_dp:N #1 } }
22669     }
22670 }

```

(End definition for `__coffin_update_corners:N`.)

`__coffin_update_poles:N` This function is called when a coffin is set, and updates the poles to reflect the nature of size of the box. Thus this function only alters poles where the default position is dependent on the size of the box. It also does not set poles which are relevant only to vertical coffins.

```

22671 \cs_new_protected:Npn \__coffin_update_poles:N #1
22672 {
22673   \prop_put:cnx { l__coffin_poles_ \__coffin_to_value:N #1 _prop } { hc }
22674   {
22675     { \dim_eval:n { 0.5 \box_wd:N #1 } }
22676     { 0pt } { 0pt } { 1000pt }
22677   }
22678   \prop_put:cnx { l__coffin_poles_ \__coffin_to_value:N #1 _prop } { r }
22679   {
22680     { \dim_eval:n { \box_wd:N #1 } }
22681     { 0pt } { 0pt } { 1000pt }
22682   }
22683   \prop_put:cnx { l__coffin_poles_ \__coffin_to_value:N #1 _prop } { vc }
22684   {
22685     { 0pt }
22686     { \dim_eval:n { ( \box_ht:N #1 - \box_dp:N #1 ) / 2 } }
22687     { 1000pt }
22688     { 0pt }
22689   }
22690   \prop_put:cnx { l__coffin_poles_ \__coffin_to_value:N #1 _prop } { t }
22691   {
22692     { 0pt }
22693     { \dim_eval:n { \box_ht:N #1 } }
22694     { 1000pt }
22695     { 0pt }
22696   }
22697   \prop_put:cnx { l__coffin_poles_ \__coffin_to_value:N #1 _prop } { b }
22698   {
22699     { 0pt }
22700     { \dim_eval:n { -\box_dp:N #1 } }
22701     { 1000pt }
22702     { 0pt }
22703   }
22704 }

```

(End definition for `__coffin_update_poles:N`.)

39.5 Coffins: calculation of pole intersections

`__coffin_calculate_intersection:Nnn` The lead off in finding intersections is to recover the two poles and then hand off to the auxiliary for the actual calculation. There may of course not be an intersection, for which an error trap is needed.

`__coffin_calculate_intersection:nnnnnnnn`
`__coffin_calculate_intersection_aux:nnnnnnN`

```

22705 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
22706 {
22707   \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
22708   \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
22709   \bool_set_false:N \l__coffin_error_bool
22710   \exp_last_two_unbraced:Noo
22711   \__coffin_calculate_intersection:nnnnnnnn

```



```

22712         \l__coffin_pole_a_tl \l__coffin_pole_b_tl
22713     \bool_if:NT \l__coffin_error_bool
22714     {
22715         \__kernel_msg_error:nn { kernel } { no-pole-intersection }
22716         \dim_zero:N \l__coffin_x_dim
22717         \dim_zero:N \l__coffin_y_dim
22718     }
22719 }

```

The two poles passed here each have four values (as dimensions), (a, b, c, d) and (a', b', c', d') . These are arguments 1–4 and 5–8, respectively. In both cases a and b are the co-ordinates of a point on the pole and c and d define the direction of the pole. Finding the intersection depends on the directions of the poles, which are given by d/c and d'/c' . However, if one of the poles is either horizontal or vertical then one or more of c, d, c' and d' are zero and a special case is needed.

```

22720 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
22721     #1#2#3#4#5#6#7#8
22722     {
22723         \dim_compare:nNnTF {#3} = { \c_zero_dim }

```

The case where the first pole is vertical. So the x -component of the interaction is at a . There is then a test on the second pole: if it is also vertical then there is an error.

```

22724     {
22725         \dim_set:Nn \l__coffin_x_dim {#1}
22726         \dim_compare:nNnTF {#7} = \c_zero_dim
22727         { \bool_set_true:N \l__coffin_error_bool }

```

The second pole may still be horizontal, in which case the y -component of the intersection is b' . If not,

$$y = \frac{d'}{c'} (x - a') + b'$$

with the x -component already known to be #1. This calculation is done as a generalised auxiliary.

```

22728     {
22729         \dim_compare:nNnTF {#8} = \c_zero_dim
22730         { \dim_set:Nn \l__coffin_y_dim {#6} }
22731         {
22732             \__coffin_calculate_intersection_aux:nnnnnnN
22733             {#1} {#5} {#6} {#7} {#8} \l__coffin_y_dim
22734         }
22735     }
22736 }

```

If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the x - and y -components interchanged.

```

22737     {
22738         \dim_compare:nNnTF {#4} = \c_zero_dim
22739         {
22740             \dim_set:Nn \l__coffin_y_dim {#2}
22741             \dim_compare:nNnTF {#8} = { \c_zero_dim }
22742             { \bool_set_true:N \l__coffin_error_bool }
22743             {
22744                 \dim_compare:nNnTF {#7} = \c_zero_dim
22745                 { \dim_set:Nn \l__coffin_x_dim {#5} }

```

The formula for the case where the second pole is neither horizontal nor vertical is

$$x = \frac{c'}{d'}(y - b') + a'$$

which is again handled by the same auxiliary.

```

22746         {
22747             \__coffin_calculate_intersection_aux:nnnnnN
22748             {#2} {#6} {#5} {#8} {#7} \l__coffin_x_dim
22749         }
22750     }
22751 }

```

The first pole is neither horizontal nor vertical. This still leaves the second pole, which may be a special case. For those possibilities, the calculations are the same as above with the first and second poles interchanged.

```

22752     {
22753         \dim_compare:nNnTF {#7} = \c_zero_dim
22754         {
22755             \dim_set:Nn \l__coffin_x_dim {#5}
22756             \__coffin_calculate_intersection_aux:nnnnnN
22757             {#5} {#1} {#2} {#3} {#4} \l__coffin_y_dim
22758         }
22759         {
22760             \dim_compare:nNnTF {#8} = \c_zero_dim
22761             {
22762                 \dim_set:Nn \l__coffin_y_dim {#6}
22763                 \__coffin_calculate_intersection_aux:nnnnnN
22764                 {#6} {#2} {#1} {#4} {#3} \l__coffin_x_dim
22765             }

```

If none of the special cases apply then there is still a need to check that there is a unique intersection between the two pole. This is the case if they have different slopes.

```

22766         {
22767             \fp_set:Nn \l__coffin_slope_x_fp
22768             { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
22769             \fp_set:Nn \l__coffin_slope_y_fp
22770             { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
22771             \fp_compare:nNnTF
22772             \l__coffin_slope_x_fp = \l__coffin_slope_y_fp
22773             { \bool_set_true:N \l__coffin_error_bool }

```

All of the tests pass, so there is the full complexity of the calculation:

$$x = \frac{a(d/c) - a'(d'/c') - b + b'}{(d/c) - (d'/c')}$$

and noting that the two ratios are already worked out from the test just performed. There is quite a bit of shuffling from dimensions to floating points in order to do the work. The y -values is then worked out using the standard auxiliary starting from the x -position.

```

22774         {
22775             \dim_set:Nn \l__coffin_x_dim
22776             {
22777                 \fp_to_dim:n

```

```

22778         {
22779             (
22780                 \dim_to_fp:n {#1} *
22781                 \l__coffin_slope_x_fp
22782             - ( \dim_to_fp:n {#5} *
22783                 \l__coffin_slope_y_fp )
22784             - \dim_to_fp:n {#2}
22785             + \dim_to_fp:n {#6}
22786         )
22787         /
22788         (
22789             \l__coffin_slope_x_fp -
22790             \l__coffin_slope_y_fp
22791         )
22792     }
22793 }
22794 \__coffin_calculate_intersection_aux:nnnnnN
22795 { \l__coffin_x_dim }
22796 {#5} {#6} {#8} {#7} \l__coffin_y_dim
22797 }
22798 }
22799 }
22800 }
22801 }
22802 }

```

The formula for finding the intersection point is in most cases the same. The formula here is

$$\#6 = \#4 \cdot \left(\frac{\#1 - \#2}{\#5} \right) \#3$$

Thus #4 and #5 should be the directions of the pole while #2 and #3 are co-ordinates.

```

22803 \cs_new_protected:Npn \__coffin_calculate_intersection_aux:nnnnnN
22804     #1#2#3#4#5#6
22805     {
22806         \dim_set:Nn #6
22807         {
22808             \fp_to_dim:n
22809             {
22810                 \dim_to_fp:n {#4} *
22811                 ( \dim_to_fp:n {#1} - \dim_to_fp:n {#2} ) /
22812                 \dim_to_fp:n {#5}
22813                 + \dim_to_fp:n {#3}
22814             }
22815         }
22816     }

```

(End definition for __coffin_calculate_intersection:Nnn, __coffin_calculate_intersection:nnnnnnnn, and __coffin_calculate_intersection_aux:nnnnnN.)

39.6 Aligning and typesetting of coffins

```

\coffin_join:NnnNnnnn
\coffin_join:cnnNnnnn
\coffin_join:Nnncnnnn
\coffin_join:cnnncnnnn

```

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which

has all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```

22817 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
22818 {
22819   \__coffin_align:NnnNnnnnN
22820   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin

```

Correct the placement of the reference point. If the x -offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which would show up if it is wider than the sum of the x -offset and the width of the second box. So a second kern may be needed.

```

22821   \hbox_set:Nn \l__coffin_aligned_coffin
22822   {
22823     \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
22824     { \tex_kern:D -\l__coffin_offset_x_dim }
22825     \hbox_unpack:N \l__coffin_aligned_coffin
22826     \dim_set:Nn \l__coffin_internal_dim
22827     { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
22828     \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
22829     { \tex_kern:D -\l__coffin_internal_dim }
22830   }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

22831   \__coffin_reset_structure:N \l__coffin_aligned_coffin
22832   \prop_clear:c
22833   {
22834     l__coffin_corners_
22835     \__coffin_to_value:N \l__coffin_aligned_coffin
22836     _prop
22837   }
22838   \__coffin_update_poles:N \l__coffin_aligned_coffin

```

The structures of the parent coffins are now transferred to the new coffin, which requires that the appropriate offsets are applied. That then depends on whether any shift was needed.

```

22839   \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
22840   {
22841     \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
22842     \__coffin_offset_poles:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
22843     \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
22844     \__coffin_offset_corners:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
22845   }
22846   {
22847     \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
22848     \__coffin_offset_poles:Nnn #4
22849     { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
22850     \__coffin_offset_corners:Nnn #1 { Opt } { Opt }
22851     \__coffin_offset_corners:Nnn #4
22852     { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
22853   }
22854   \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
22855   \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
22856 }
22857 \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cncnc }

```

(End definition for `\coffin_join:NnnNnnnn`. This function is documented on page 228.)

`\coffin_attach:NnnNnnnn` A more simple version of the above, as it simply uses the size of the first coffin for the
`\coffin_attach:cnmNnnnn` new one. This means that the work here is rather simplified compared to the above code.
`\coffin_attach:Nnncnnnn` The function used when marking a position is hear also as it is similar but without the
`\coffin_attach:cnncnnnn` structure updates.

```

\coffin_attach_mark:NnnNnnnn 22858 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
22859 {
22860   \__coffin_align:NnnNnnnnN
22861   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
22862   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
22863   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
22864   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
22865   \__coffin_reset_structure:N \l__coffin_aligned_coffin
22866   \prop_set_eq:cc
22867   {
22868     l__coffin_corners_
22869     \__coffin_to_value:N \l__coffin_aligned_coffin _prop
22870   }
22871   { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
22872   \__coffin_update_poles:N \l__coffin_aligned_coffin
22873   \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
22874   \__coffin_offset_poles:Nnn #4
22875   { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
22876   \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
22877   \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
22878 }
22879 \cs_new_protected:Npn \coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
22880 {
22881   \__coffin_align:NnnNnnnnN
22882   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
22883   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
22884   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
22885   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
22886   \box_set_eq:NN #1 \l__coffin_aligned_coffin
22887 }
22888 \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }

```

(End definition for `\coffin_attach:NnnNnnnn` and `\coffin_attach_mark:NnnNnnnn`. These functions are documented on page 227.)

`__coffin_align:NnnNnnnnN` The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the ‘primed’ storage area to be used for the second coffin. The ‘real’ box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result depends on how the bounding box is being handled.

```

22889 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
22890 {
22891   \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
22892   \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
22893   \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }

```

```

22894 \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
22895 \dim_set:Nn \l__coffin_offset_x_dim
22896 { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
22897 \dim_set:Nn \l__coffin_offset_y_dim
22898 { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
22899 \hbox_set:Nn \l__coffin_aligned_internal_coffin
22900 {
22901   \box_use:N #1
22902   \tex_kern:D -\box_wd:N #1
22903   \tex_kern:D \l__coffin_offset_x_dim
22904   \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
22905 }
22906 \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
22907 }

```

(End definition for __coffin_align:NnnNnnnnN.)

__coffin_offset_poles:Nnn
 __coffin_offset_pole:Nnnnnnn

Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping to the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a - means that the structures from the parent coffins are uniquely labelled and do not depend on the order of alignment. The pay off for this is that - should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

```

22908 \cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
22909 {
22910   \prop_map_inline:cn { l__coffin_poles_ \__coffin_to_value:N #1 _prop }
22911   { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
22912 }
22913 \cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
22914 {
22915   \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
22916   \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
22917   \tl_if_in:nnTF {#2} { - }
22918   { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
22919   { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
22920   \exp_last_unbraced:NNo \__coffin_set_pole:Nnx \l__coffin_aligned_coffin
22921   { \l__coffin_internal_tl }
22922   {
22923     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
22924     {#5} {#6}
22925   }
22926 }

```

(End definition for __coffin_offset_poles:Nnn and __coffin_offset_pole:Nnnnnnn.)

__coffin_offset_corners:Nnn
 __coffin_offset_corner:Nnnnn

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.

```

22927 \cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
22928 {
22929   \prop_map_inline:cn { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
22930   { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
22931 }
22932 \cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6

```

```

22933 {
22934   \prop_put:cnx
22935   {
22936     l__coffin_corners_
22937     \__coffin_to_value:N \l__coffin_aligned_coffin _prop
22938   }
22939   { #1 - #2 }
22940   {
22941     { \dim_eval:n { #3 + #5 } }
22942     { \dim_eval:n { #4 + #6 } }
22943   }
22944 }

```

(End definition for __coffin_offset_corners:Nnn and __coffin_offset_corner:Nnnnn.)

__coffin_update_vertical_poles:NNN
 __coffin_update_T:nnnnnnnnN
 __coffin_update_B:nnnnnnnnN

The T and B poles need to be recalculated after alignment. These functions find the larger absolute value for the poles, but this is of course only logical when the poles are horizontal.

```

22945 \cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
22946 {
22947   \__coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
22948   \__coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
22949   \exp_last_two_unbraced:Noo \__coffin_update_T:nnnnnnnnN
22950   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
22951   \__coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
22952   \__coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
22953   \exp_last_two_unbraced:Noo \__coffin_update_B:nnnnnnnnN
22954   \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
22955 }
22956 \cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
22957 {
22958   \dim_compare:nNnTF {#2} < {#6}
22959   {
22960     \__coffin_set_pole:Nnx #9 { T }
22961     { { Opt } {#6} { 1000pt } { Opt } }
22962   }
22963   {
22964     \__coffin_set_pole:Nnx #9 { T }
22965     { { Opt } {#2} { 1000pt } { Opt } }
22966   }
22967 }
22968 \cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
22969 {
22970   \dim_compare:nNnTF {#2} < {#6}
22971   {
22972     \__coffin_set_pole:Nnx #9 { B }
22973     { { Opt } {#2} { 1000pt } { Opt } }
22974   }
22975   {
22976     \__coffin_set_pole:Nnx #9 { B }
22977     { { Opt } {#6} { 1000pt } { Opt } }
22978   }
22979 }

```

(End definition for `_coffin_update_vertical_poles:NNN`, `_coffin_update_T:nnnnnnnnN`, and `_coffin_update_B:nnnnnnnnN`.)

`\coffin_typeset:Nnnnn` Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

`\coffin_typeset:cnnnn`

```
22980 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
22981 {
22982   \mode_leave_vertical:
22983   \_coffin_align:NnnNnnnnN \c_empty_coffin { H } { l }
22984   #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
22985   \box_use_drop:N \l__coffin_aligned_coffin
22986 }
22987 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }
```

(End definition for `\coffin_typeset:Nnnnn`. This function is documented on page 228.)

39.7 Coffin diagnostics

`\l__coffin_display_coffin` Used for printing coffins with data structures attached.

`\l__coffin_display_coord_coffin`
`\l__coffin_display_pole_coffin`

```
22988 \coffin_new:N \l__coffin_display_coffin
22989 \coffin_new:N \l__coffin_display_coord_coffin
22990 \coffin_new:N \l__coffin_display_pole_coffin
```

(End definition for `\l__coffin_display_coffin`, `\l__coffin_display_coord_coffin`, and `\l__coffin_display_pole_coffin`.)

`\l__coffin_display_handles_prop` This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```
22991 \prop_new:N \l__coffin_display_handles_prop
22992 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
22993 { { b } { r } { -1 } { 1 } }
22994 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
22995 { { b } { hc } { 0 } { 1 } }
22996 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
22997 { { b } { l } { 1 } { 1 } }
22998 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
22999 { { vc } { r } { -1 } { 0 } }
23000 \prop_put:Nnn \l__coffin_display_handles_prop { vhc }
23001 { { vc } { hc } { 0 } { 0 } }
23002 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
23003 { { vc } { l } { 1 } { 0 } }
23004 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
23005 { { t } { r } { -1 } { -1 } }
23006 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
23007 { { t } { hc } { 0 } { -1 } }
23008 \prop_put:Nnn \l__coffin_display_handles_prop { br }
23009 { { t } { l } { 1 } { -1 } }
23010 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
23011 { { t } { r } { -1 } { -1 } }
23012 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
23013 { { t } { hc } { 0 } { -1 } }
23014 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
23015 { { t } { l } { 1 } { -1 } }
```



```

23016 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
23017 { { vc } { r } { -1 } { 1 } }
23018 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
23019 { { vc } { hc } { 0 } { 1 } }
23020 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
23021 { { vc } { l } { 1 } { 1 } }
23022 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
23023 { { b } { r } { -1 } { -1 } }
23024 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
23025 { { b } { hc } { 0 } { -1 } }
23026 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
23027 { { b } { l } { 1 } { -1 } }

```

(End definition for `\l__coffin_display_handles_prop`.)

`\l__coffin_display_offset_dim` The standard offset for the label from the handle position when displaying handles.

```

23028 \dim_new:N \l__coffin_display_offset_dim
23029 \dim_set:Nn \l__coffin_display_offset_dim { 2pt }

```

(End definition for `\l__coffin_display_offset_dim`.)

`\l__coffin_display_x_dim` `\l__coffin_display_y_dim` As the intersections of poles have to be calculated to find which ones to print, there is a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```

23030 \dim_new:N \l__coffin_display_x_dim
23031 \dim_new:N \l__coffin_display_y_dim

```

(End definition for `\l__coffin_display_x_dim` and `\l__coffin_display_y_dim`.)

`\l__coffin_display_poles_prop` A property list for printing poles: various things need to be deleted from this to get a “nice” output.

```

23032 \prop_new:N \l__coffin_display_poles_prop

```

(End definition for `\l__coffin_display_poles_prop`.)

`\l__coffin_display_font_tl` Stores the settings used to print coffin data: this keeps things flexible.

```

23033 \tl_new:N \l__coffin_display_font_tl
23034 \*initex
23035 \tl_set:Nn \l__coffin_display_font_tl { } % TODO
23036 \*initex
23037 \*package
23038 \tl_set:Nn \l__coffin_display_font_tl { \sfamily \tiny }
23039 \*package

```

(End definition for `\l__coffin_display_font_tl`.)

`\coffin_mark_handle:Nnnn` `\coffin_mark_handle:cnnn` `__coffin_mark_handle_aux:nnnnNnn` Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

```

23040 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
23041 {
23042   \hcoffin_set:Nn \l__coffin_display_pole_coffin
23043   {
23044     \*initex

```

```

23045         \hbox:n { \tex_vrule:D width 1pt height 1pt \scan_stop: } % TODO
23046 \end{initex}
23047 \begin{package}
23048     \color {#4}
23049     \rule { 1pt } { 1pt }
23050 \end{package}
23051 }
23052 \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
23053 \l__coffin_display_pole_coffin { hc } { vc } { Opt } { Opt }
23054 \hcoffin_set:Nn \l__coffin_display_coord_coffin
23055 {
23056 \begin{initex}
23057     % TODO
23058 \end{initex}
23059 \begin{package}
23060     \color {#4}
23061 \end{package}
23062 \l__coffin_display_font_tl
23063 ( \tl_to_str:n { #2 , #3 } )
23064 }
23065 \prop_get:NnN \l__coffin_display_handles_prop
23066 { #2 #3 } \l__coffin_internal_tl
23067 \quark_if_no_value:NTF \l__coffin_internal_tl
23068 {
23069     \prop_get:NnN \l__coffin_display_handles_prop
23070     { #3 #2 } \l__coffin_internal_tl
23071     \quark_if_no_value:NTF \l__coffin_internal_tl
23072     {
23073         \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
23074         \l__coffin_display_coord_coffin { l } { vc }
23075         { 1pt } { Opt }
23076     }
23077     {
23078         \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
23079         \l__coffin_internal_tl #1 {#2} {#3}
23080     }
23081 }
23082 {
23083     \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
23084     \l__coffin_internal_tl #1 {#2} {#3}
23085 }
23086 }
23087 \cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
23088 {
23089     \coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
23090     \l__coffin_display_coord_coffin {#1} {#2}
23091     { #3 \l__coffin_display_offset_dim }
23092     { #4 \l__coffin_display_offset_dim }
23093 }
23094 \cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }

```

(End definition for `\coffin_mark_handle:Nnnn` and `__coffin_mark_handle_aux:nnnnNnn`. This function is documented on page 229.)

\coffin_display_handles:Nn Printing the poles starts by removing any duplicates, for which the H poles is used as

```

\coffin_display_handles:cn
  \__coffin_display_handles_aux:nnnnnn
  \__coffin_display_handles_aux:nnnn
  \__coffin_display_attach:Nnnnn

```

the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```

23095 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
23096 {
23097   \hcoffin_set:Nn \l__coffin_display_pole_coffin
23098   {
23099     \*initex
23100     \hbox:n { \tex_vrule:D width 1pt height 1pt \scan_stop: } % TODO
23101   }
23102   \package
23103   \color {#2}
23104   \rule { 1pt } { 1pt }
23105 \endpackage
23106 }
23107 \prop_set_eq:Nc \l__coffin_display_poles_prop
23108 { \l__coffin_poles_ \__coffin_to_value:N #1 _prop }
23109 \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
23110 \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
23111 \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
23112 { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
23113 \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
23114 \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
23115 { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
23116 \coffin_set_eq:NN \l__coffin_display_coffin #1
23117 \prop_map_inline:Nn \l__coffin_display_poles_prop
23118 {
23119   \prop_remove:Nn \l__coffin_display_poles_prop {##1}
23120   \__coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
23121 }
23122 \box_use_drop:N \l__coffin_display_coffin
23123 }

```

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.

```

23124 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
23125 {
23126   \prop_map_inline:Nn \l__coffin_display_poles_prop
23127   {
23128     \bool_set_false:N \l__coffin_error_bool
23129     \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
23130     \bool_if:NF \l__coffin_error_bool
23131     {
23132       \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
23133       \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
23134       \__coffin_display_attach:Nnnnn
23135       \l__coffin_display_pole_coffin { hc } { vc }
23136       { Opt } { Opt }
23137       \hcoffin_set:Nn \l__coffin_display_coord_coffin
23138       {
23139         \*initex
23140         % TODO
23141       }

```

```

23142 \*package>
23143         \color {#6}
23144 \package>
23145         \l__coffin_display_font_tl
23146         ( \tl_to_str:n { #1 , ##1 } )
23147     }
23148     \prop_get:NnN \l__coffin_display_handles_prop
23149     { #1 ##1 } \l__coffin_internal_tl
23150     \quark_if_no_value:NTF \l__coffin_internal_tl
23151     {
23152         \prop_get:NnN \l__coffin_display_handles_prop
23153         { ##1 #1 } \l__coffin_internal_tl
23154         \quark_if_no_value:NTF \l__coffin_internal_tl
23155         {
23156             \__coffin_display_attach:Nnnnn
23157             \l__coffin_display_coord_coffin { 1 } { vc }
23158             { 1pt } { 0pt }
23159         }
23160         {
23161             \exp_last_unbraced:No
23162             \__coffin_display_handles_aux:nnnn
23163             \l__coffin_internal_tl
23164         }
23165     }
23166     {
23167         \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
23168         \l__coffin_internal_tl
23169     }
23170 }
23171 }
23172 }
23173 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
23174 {
23175     \__coffin_display_attach:Nnnnn
23176     \l__coffin_display_coord_coffin {#1} {#2}
23177     { #3 \l__coffin_display_offset_dim }
23178     { #4 \l__coffin_display_offset_dim }
23179 }
23180 \cs_generate_variant:Nn \coffin_display_handles:Nn { c }

```

This is a dedicated version of `\coffin_attach:NnnNnnnn` with a hard-wired first coffin. As the intersection is already known and stored for the display coffin the code simply uses it directly, with no calculation.

```

23181 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
23182 {
23183     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
23184     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
23185     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
23186     \dim_set:Nn \l__coffin_offset_x_dim
23187     { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
23188     \dim_set:Nn \l__coffin_offset_y_dim
23189     { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
23190     \hbox_set:Nn \l__coffin_aligned_coffin
23191     {

```

```

23192         \box_use:N \l__coffin_display_coffin
23193         \tex_kern:D -\box_wd:N \l__coffin_display_coffin
23194         \tex_kern:D \l__coffin_offset_x_dim
23195         \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
23196     }
23197     \box_set_ht:Nn \l__coffin_aligned_coffin
23198     { \box_ht:N \l__coffin_display_coffin }
23199     \box_set_dp:Nn \l__coffin_aligned_coffin
23200     { \box_dp:N \l__coffin_display_coffin }
23201     \box_set_wd:Nn \l__coffin_aligned_coffin
23202     { \box_wd:N \l__coffin_display_coffin }
23203     \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
23204 }

```

(End definition for `\coffin_display_handles:Nn` and others. This function is documented on page 228.)

```

\coffin_show_structure:N
\coffin_show_structure:c
\coffin_log_structure:N
\coffin_log_structure:c
\__coffin_show_structure:NN

```

For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

```

23205 \cs_new_protected:Npn \coffin_show_structure:N
23206 { \__coffin_show_structure:NN \msg_show:nnxxxx }
23207 \cs_generate_variant:Nn \coffin_show_structure:N { c }
23208 \cs_new_protected:Npn \coffin_log_structure:N
23209 { \__coffin_show_structure:NN \msg_log:nnxxxx }
23210 \cs_generate_variant:Nn \coffin_log_structure:N { c }
23211 \cs_new_protected:Npn \__coffin_show_structure:NN #1#2
23212 {
23213     \__coffin_if_exist:NT #2
23214     {
23215         #1 { LaTeX / kernel } { show-coffin }
23216         { \token_to_str:N #2 }
23217         {
23218             \iow_newline: >~ ht ~~~ \dim_eval:n { \coffin_ht:N #2 }
23219             \iow_newline: >~ dp ~~~ \dim_eval:n { \coffin_dp:N #2 }
23220             \iow_newline: >~ wd ~~~ \dim_eval:n { \coffin_wd:N #2 }
23221         }
23222         {
23223             \prop_map_function:cN
23224             { l__coffin_poles_ \__coffin_to_value:N #2 _prop }
23225             \msg_show_item_unbraced:nn
23226         }
23227     } }
23228 }
23229 }

```

(End definition for `\coffin_show_structure:N`, `\coffin_log_structure:N`, and `__coffin_show_structure:NN`. These functions are documented on page 229.)

39.8 Messages

```

23230 \__kernel_msg_new:nnnn { kernel } { no-pole-intersection }
23231 { No~intersection~between~coffin~poles. }
23232 {
23233     LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
23234     but~they~do~not~have~a~unique~meeting~point:~

```

```

23235     the~value~(0pt,~0pt)~will~be~used.
23236   }
23237   \__kernel_msg_new:nnnn { kernel } { unknown-coffin }
23238   { Unknown~coffin~'#1'. }
23239   { The~coffin~'#1'~was~never~defined. }
23240   \__kernel_msg_new:nnnn { kernel } { unknown-coffin-pole }
23241   { Pole~'#1'~unknown~for~coffin~'#2'. }
23242   {
23243     LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
23244     but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
23245   }
23246   \__kernel_msg_new:nnn { kernel } { show-coffin }
23247   {
23248     Size~of~coffin~#1 : #2 \\
23249     Poles~of~coffin~#1 : #3 .
23250   }
23251 </initex | package>

```

40 l3color-base Implementation

```

23252 (*initex | package)
23253 <@@=color>

```

`\l__color_current_tl` The color currently active for foreground (text, *etc.*) material. This is stored in the form of a color model followed by one or more values. There are four pre-defined models, three of which take numerical values in the range [0, 1]:

- `gray` `<gray>` Grayscale color with the `<gray>` value running from 0 (fully black) to 1 (fully white)
- `cmyk` `<cyan>` `<magenta>` `<yellow>` `<black>`
- `rgb` `<red>` `<green>` `<blue>`

Notice that the value are separated by spaces. There is a fourth pre-defined model using a string value and a numerical one:

- `spot` `<name>` `<tint>` A pre-defined spot color, where the `<name>` should be a pre-defined string color name and the `<tint>` should be in the range [0, 1].

Additional models may be created to allow mixing of spot colors. The number of data entries these require will depend on the number of colors to be mixed.

T_EXhackers note: The content of `\l__color_current_tl` is space-separated as this allows it to be used directly in specials in many common cases. This internal representation is close to that used by the `dvips` program.

(End definition for `\l__color_current_tl`.)

`\color_group_begin:` Grouping for color is almost the same as using the basic `\group_begin:` and `\group_end:` functions. However, in vertical mode the end-of-group needs a `\par`, which in horizontal mode does nothing.

```

23254 \cs_new_eq:NN \color_group_begin: \group_begin:
23255 \cs_new_protected:Npn \color_group_end:

```

```

23256 {
23257     \par
23258     \group_end:
23259 }

```

(End definition for `\color_group_begin:` and `\color_group_end:`. These functions are documented on page 230.)

`\color_ensure_current:` A driver-independent wrapper for setting the foreground color to the current color “now”.

```

23260 \cs_new_protected:Npn \color_ensure_current:
23261 {
23262     <*package>
23263     \driver_color_pickup:N \l__color_current_tl
23264 </package>
23265     \__color_select:V \l__color_current_tl
23266 }

```

(End definition for `\color_ensure_current:`. This function is documented on page 230.)

`__color_select:n` Take an internal color specification and pass it to the driver. This code is needed to ensure the current color but will also be used by the higher-level experimental material.

```

\__color_select:V
\__color_select:w
\__color_select_cmyk:w
\__color_select_gray:w
\__color_select_rgb:w
\__color_select_spot:w
23267 \cs_new_protected:Npn \__color_select:n #1
23268 { \__color_select:w #1 \q_stop }
23269 \cs_generate_variant:Nn \__color_select:n { V }
23270 \cs_new_protected:Npn \__color_select:w #1 ~ #2 \q_stop
23271 { \use:c { __color_select_ #1 :w } #2 \q_stop }
23272 \cs_new_protected:Npn \__color_select_cmyk:w #1 ~ #2 ~ #3 ~ #4 \q_stop
23273 { \driver_color_cmyk:nnnn {#1} {#2} {#3} {#4} }
23274 \cs_new_protected:Npn \__color_select_gray:w #1 \q_stop
23275 { \driver_color_gray:n {#1} }
23276 \cs_new_protected:Npn \__color_select_rgb:w #1 ~ #2 ~ #3 \q_stop
23277 { \driver_color_rgb:nnn {#1} {#2} {#3} }
23278 \cs_new_protected:Npn \__color_select_spot:w #1 ~ #2 \q_stop
23279 { \driver_color_spot:nn {#1} {#2} }

```

(End definition for `__color_select:n` and others.)

`\l__color_current_tl` As the setting data is used only for specials, and those are always space-separated, it makes most sense to hold the internal information in that form.

```

23280 \tl_new:N \l__color_current_tl
23281 \tl_set:Nn \l__color_current_tl { gray~0 }

```

(End definition for `\l__color_current_tl`.)

```

23282 </initex | package>

```

41 l3luatex implementation

```

23283 <*initex | package>

```

41.1 Breaking out to Lua

```

23284 <*tex>
23285 <@@=luatex>

```

```

\__luatex_escape_x:n  Copies of primitives.
\__luatex_now_x:n      23286 \cs_new_eq:NN \__luatex_escape_x:n \luatex_luaescapestring:D
\__luatex_shipout_x:n  23287 \cs_new_eq:NN \__luatex_now_x:n \luatex_directlua:D
                        23288 \cs_new_eq:NN \__luatex_shipout_x:n \luatex_latelua:D

(End definition for \__luatex_escape_x:n, \__luatex_now_x:n, and \__luatex_shipout_x:n.)

These functions are set up in l3str for bootstrapping: we want to replace them with
a “proper” version at this stage, so clean up.

23289 \cs_undefine:N \lua_escape_x:n
23290 \cs_undefine:N \lua_now_x:n

\lua_now_x:n  Wrappers around the primitives. As with engines other than LuaTeX these have to be
\lua_now_x:n macros, we give them the same status in all cases. When LuaTeX is not in use, simply
\lua_shipout_x:n give an error message/
\lua_shipout_x:n
\lua_escape_x:n 23291 \cs_new:Npn \lua_now_x:n #1 { \__luatex_now_x:n {#1} }
\lua_escape_x:n 23292 \cs_new:Npn \lua_now_x:n #1 { \lua_now_x:n { \exp_not:n {#1} } }
23293 \cs_new_protected:Npn \lua_shipout_x:n #1 { \__luatex_shipout_x:n {#1} }
23294 \cs_new_protected:Npn \lua_shipout_x:n #1
23295 { \lua_shipout_x:n { \exp_not:n {#1} } }
23296 \cs_new:Npn \lua_escape_x:n #1 { \__luatex_escape_x:n {#1} }
23297 \cs_new:Npn \lua_escape_x:n #1 { \lua_escape_x:n { \exp_not:n {#1} } }
23298 \sys_if_engine luatex:F
23299 {
23300   \clist_map_inline:nn
23301   {
23302     \use_x:n ,
23303     \lua_escape_x:n , \lua_escape:n ,
23304     \lua_now_x:n , \lua_now:n
23305   }
23306   {
23307     \cs_set:Npn #1 ##1
23308     {
23309       \__kernel_msg_expandable_error:nnn
23310       { kernel } { luatex-required } { #1 }
23311     }
23312   }
23313   \clist_map_inline:nn
23314   { \lua_shipout_x:n , \lua_shipout:n }
23315   {
23316     \cs_set_protected:Npn #1 ##1
23317     {
23318       \__kernel_msg_error:nnn
23319       { kernel } { luatex-required } { #1 }
23320     }
23321   }
23322 }

```

(End definition for \lua_now_x:n and others. These functions are documented on page 231.)

41.2 Messages

```

23323 \__kernel_msg_new:nnnn { kernel } { luatex-required }
23324 { LuaTeX~engine~not~in~use!~Ignoring~#1. }
23325 {

```



```

23326     The~feature~you~are~using~is~only~available~
23327     with~the~LuaTeX~engine.~LaTeX3~ignored~'~#1'~.
23328 }
23329 </tex>

```

41.3 Lua functions for internal use

```

23330 (*lua)

```

Most of the emulation of pdfTeX here is based heavily on Heiko Oberdiek's `pdfTeXcmds` package.

l3kernel Create a table for the kernel's own use.

```

23331 l3kernel = l3kernel or { }

```

(End definition for l3kernel. This function is documented on page 232.)

Local copies of global tables.

```

23332 local io      = io
23333 local kpse    = kpse
23334 local lfs     = lfs
23335 local math    = math
23336 local md5     = md5
23337 local os      = os
23338 local string  = string
23339 local tex     = tex
23340 local unicode = unicode

```

Local copies of standard functions.

```

23341 local abs      = math.abs
23342 local byte     = string.byte
23343 local floor    = math.floor
23344 local format   = string.format
23345 local gsub     = string.gsub
23346 local kpse_find = kpse.find_file
23347 local lfs_attr = lfs.attributes
23348 local md5_sum  = md5.sum
23349 local open     = io.open
23350 local os_date  = os.date
23351 local setcatcode = tex.setcatcode
23352 local str_format = string.format
23353 local sprint   = tex.sprint
23354 local write    = tex.write
23355 local utf8_char = unicode.utf8.char

```

escapehex An internal auxiliary to convert a string to the matching hex escape. This works on a byte basis: extension to handled UTF-8 input is covered in `pdfTeXcmds` but is not currently required here.

```

23356 local function escapehex(str)
23357     write((gsub(str, ".",
23358         function (ch) return format("%02X", byte(ch)) end)))
23359 end

```

(End definition for escapehex.)

l3kernel.charcat Creating arbitrary chars needs a category code table. As set up here, one may have been assigned earlier (see `l3bootstrap`) or a hard-coded one is used. The latter is intended for format mode and should be adjusted to match an eventual allocator.

```

23360 local charcat_table = l3kernel.charcat_table or 1
23361 local function charcat(charcode, catcode)
23362     setcatcode(charcat_table, charcode, catcode)
23363     sprint(charcat_table, utf8_char(charcode))
23364 end
23365 l3kernel.charcat = charcat

```

(End definition for `l3kernel.charcat`. This function is documented on page 232.)

l3kernel.filemdfivesum Read an entire file and hash it: the hash function itself is a built-in. As Lua is byte-based there is no work needed here in terms of UTF-8 (see `pdfexcmds` and how it handles strings that have passed through `LuaTeX`). The file is read in binary mode so that no line ending normalisation occurs.

```

23366 local function filemdfivesum(name)
23367     local file = kpse_find(name, "tex", true)
23368     if file then
23369         local f = open(file, "rb")
23370         if f then
23371             local data = f:read("*a")
23372             escapehex(md5_sum(data))
23373             f:close()
23374         end
23375     end
23376 end
23377 l3kernel.filemdfivesum = filemdfivesum

```

(End definition for `l3kernel.filemdfivesum`. This function is documented on page 232.)

l3kernel.filemoddate See procedure `makepdftime` in `utils.c` of `pdfTeX`.

```

23378 local function filemoddate(name)
23379     local file = kpse_find(name, "tex", true)
23380     if file then
23381         local date = lfs_attr(file, "modification")
23382         if date then
23383             local d = os_date("!*t", date)
23384             if d.sec >= 60 then
23385                 d.sec = 59
23386             end
23387             local u = os_date("!*t", date)
23388             local off = 60 * (d.hour - u.hour) + d.min - u.min
23389             if d.year ~= u.year then
23390                 if d.year > u.year then
23391                     off = off + 1440
23392                 else
23393                     off = off - 1440
23394                 end
23395             elseif d.yday ~= u.yday then
23396                 if d.yday > u.yday then
23397                     off = off + 1440
23398                 else
23399                     off = off - 1440
23399

```

```

23400         end
23401     end
23402     local timezone
23403     if off == 0 then
23404         timezone = "Z"
23405     else
23406         local hours = floor(off / 60)
23407         local mins = abs(off - hours * 60)
23408         timezone = str_format("%+03d", hours)
23409         .. "' ' " .. str_format("%02d", mins) .. "' ' "
23410     end
23411     write("D:"
23412         .. str_format("%04d", d.year)
23413         .. str_format("%02d", d.month)
23414         .. str_format("%02d", d.day)
23415         .. str_format("%02d", d.hour)
23416         .. str_format("%02d", d.min)
23417         .. str_format("%02d", d.sec)
23418         .. timezone)
23419     end
23420 end
23421 end
23422 l3kernel.filemoddate = filemoddate

```

(End definition for `l3kernel.filemoddate`. This function is documented on page 232.)

l3kernel.filesize A simple disk lookup.

```

23423 local function filesize(name)
23424     local file = kpse_find(name, "tex", true)
23425     if file then
23426         local size = lfs_attr(file, "size")
23427         if size then
23428             write(size)
23429         end
23430     end
23431 end
23432 l3kernel.filesize = filesize

```

(End definition for `l3kernel.filesize`. This function is documented on page 232.)

l3kernel.strcmp String comparison which gives the same results as pdfTeX's `\pdfstrcmp`, although the ordering should likely not be relied upon!

```

23433 local function strcmp(A, B)
23434     if A == B then
23435         write("0")
23436     elseif A < B then
23437         write("-1")
23438     else
23439         write("1")
23440     end
23441 end
23442 l3kernel.strcmp = strcmp

```

(End definition for `l3kernel.strcmp`. This function is documented on page 232.)

41.4 Generic Lua and font support

```
23443 <*initex>
23444 <@@=alloc>
```

A small amount of generic code is used by almost all LuaTeX material so needs to be loaded by the format.

```
23445 attribute_count_name = "g__alloc_attribute_int"
23446 bytecode_count_name  = "g__alloc_bytecode_int"
23447 chunkname_count_name  = "g__alloc_chunkname_int"
23448 whatsit_count_name    = "g__alloc_whatsit_int"
23449 require("lualatex")
```

With the above available the font loader code used by plain TeX and L^AT_EX 2_ε when used with LuaTeX can be loaded here. This is thus being treated more-or-less as part of the engine itself.

```
23450 require("luaotfload-main")
23451 local _void = luaotfload.main()
23452 </initex>
23453 </lua>
23454 </initex | package>
```

42 l3unicode implementation

```
23455 <*initex | package>
23456 <@@=char>
```

Case changing both for strings and “text” requires data from the Unicode Consortium. Some of this is build in to the format (as `\lccode` and `\uccode` values) but this covers only the simple one-to-one situations and does not fully handle for example case folding.

As only the data needs to remain at the end of this process, everything is set up inside a group. The only thing that is outside is creating a stream: they are global anyway and it is best to force a stream for all engines.

```
23457 \ior_new:N \g__char_data_ior
23458 \bool_lazy_or:nnTF { \sys_if_engine_luatex_p: } { \sys_if_engine_xetex_p: }
23459 {
23460   \group_begin:
```

Set up a private copy of the char-generation primitive.

```
23461   \cs_set_eq:NN \__char_generate:w \utex_char:D
```

Parse the main Unicode data file for title case exceptions (the one-to-one lower and upper case mappings it contains are all be covered by the TeX data). There are no comments in the main data file so this can be done using a standard mapping and no checks.

```
23462   \ior_open:Nn \g__char_data_ior { UnicodeData.txt }
23463   \cs_set_protected:Npn \__char_data_auxi:w
23464     #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
23465     { \__char_data_auxii:w #1 ; }
23466   \cs_set_protected:Npn \__char_data_auxii:w
23467     #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 \q_stop
23468     {
23469       \tl_if_blank:nF {#7}
23470       {
```

```

23471         \str_if_eq:nnF {#5} {#7}
23472         {
23473             \tl_const:cx
23474             { c__char_mixed_case_ \__char_generate:w "#1 _tl }
23475             { \__char_generate:w "#7 }
23476         }
23477     }
23478 }
23479 \ior_map_inline:Nn \g__char_data_ior
23480 {
23481     \tl_if_blank:nF {#1}
23482     { \__char_data_auxi:w #1 \q_stop }
23483 }
23484 \ior_close:N \g__char_data_ior

```

The other data files all use C-style comments so we have to worry about # tokens (and reading as strings). The set up for case folding is in two parts. For the basic (core) mappings, folding is the same as lower casing in most positions so only store the differences. For the more complex foldings, always store the result, splitting up the two or three code points in the input as required.

```

23485     \ior_open:Nn \g__char_data_ior { CaseFolding.txt }
23486     \cs_set_protected:Npn \__char_data_auxi:w #1 ;~ #2 ;~ #3 ; #4 \q_stop
23487     {
23488         \str_if_eq:nnTF {#2} { C }
23489         {
23490             \int_compare:nNnF
23491             { \char_value_lccode:n {"#1} } = {"#3}
23492             {
23493                 \tl_const:cx
23494                 { c__char_fold_case_ \__char_generate:w "#1 _tl }
23495                 { \__char_generate:w "#3 }
23496             }
23497         }
23498         {
23499             \str_if_eq:nnT {#2} { F }
23500             { \__char_data_auxii:w #1 ~ #3 ~ \q_stop }
23501         }
23502     }
23503     \cs_set_protected:Npn \__char_data_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
23504     {
23505         \tl_const:cx { c__char_fold_case_ \__char_generate:w "#1 _tl }
23506         {
23507             \__char_generate:w "#2 \exp_stop_f:
23508             \__char_generate:w "#3 \exp_stop_f:
23509             \tl_if_blank:nF {#4}
23510             { \__char_generate:w "#4 }
23511         }
23512     }
23513     \ior_str_map_inline:Nn \g__char_data_ior
23514     {
23515         \tl_if_blank:nF {#1}
23516         {
23517             \str_if_eq_x:nnF { \tl_head:n {#1} } { \c_hash_str }
23518             { \__char_data_auxi:w #1 \q_stop }

```

```

23519         }
23520     }
23521     \ior_close:N \g__char_data_ior

```

For upper and lower casing special situations, there is a bit more to do as we also have title casing to consider, plus we need to stop part-way through the file.

```

23522     \ior_open:Nn \g__char_data_ior { SpecialCasing.txt }
23523     \cs_set_protected:Npn \__char_data_auxi:w
23524         #1 ;~ #2 ;~ #3 ;~ #4 ; #5 \q_stop
23525     {
23526         \use:n { \__char_data_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
23527         \use:n { \__char_data_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
23528         \str_if_eq:nnF {#3} {#4}
23529         { \use:n { \__char_data_auxii:w #1 ~ mixed ~ #3 ~ } ~ \q_stop }
23530     }
23531     \cs_set_protected:Npn \__char_data_auxii:w
23532         #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
23533     {
23534         \tl_if_empty:nF {#4}
23535         {
23536             \tl_const:cx { c__char_ #2 _case_ \__char_generate:w "#1 _tl }
23537             {
23538                 \__char_generate:w "#3 \exp_stop_f:
23539                 \__char_generate:w "#4 \exp_stop_f:
23540                 \tl_if_blank:nF {#5}
23541                 { \__char_generate:w "#5 }
23542             }
23543         }
23544     }
23545     \ior_str_map_inline:Nn \g__char_data_ior
23546     {
23547         \tl_if_blank:nF {#1}
23548         {
23549             \str_if_eq:x:nnTF { \tl_head:n {#1} } { { \c_hash_str }
23550             {
23551                 \str_if_eq:x:nnT
23552                 {#1}
23553                 { \c_hash_str \c_space_tl Conditional~Mappings }
23554                 { \ior_map_break: }
23555             }
23556             { \__char_data_auxi:w #1 \q_stop }
23557         }
23558     }
23559     \ior_close:N \g__char_data_ior
23560     \group_end:
23561 }

```

For the 8-bit engines, the above is skipped but there is still some set up required. As case changing can only be applied to bytes, and they have to be in the ASCII range, we define a series of data stores to represent them, and the data are used such that only these are ever case-changed. We do open and close one file to force allocation of a read: this keeps all engines in line.

```

23562 {
23563     \group_begin:
23564     \cs_set_protected:Npn \__char_tmp:NN #1#2

```

```

23565     {
23566         \quark_if_recursion_tail_stop:N #2
23567         \tl_const:cn { c__char_upper_case_ #2 _tl } {#1}
23568         \tl_const:cn { c__char_lower_case_ #1 _tl } {#2}
23569         \tl_const:cn { c__char_fold_case_ #1 _tl } {#2}
23570         \__char_tmp:NN
23571     }
23572     \__char_tmp:NN
23573     AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
23574     ? \q_recursion_tail \q_recursion_stop
23575     \ior_open:Nn \g__char_data_ior { UnicodeData.txt }
23576     \ior_close:N \g__char_data_ior
23577     \group_end:
23578 }
23579 </initex | package>

```

43 l3candidates Implementation

```

23580 <*initex | package>

```

43.1 Additions to l3basics

\mode_leave_vertical: The approach here is different to that used by L^AT_EX_{2 ϵ} or plain T_EX, which unbox a void box to force horizontal mode. That inserts the `\everypar` tokens *before* the re-inserted unboxing tokens. The approach here uses either the `\quitvmode` primitive or the equivalent protected macro. In vertical mode, the `\indent` primitive is inserted: this will switch to horizontal mode and insert `\everypar` tokens and nothing else. Unlike the L^AT_EX_{2 ϵ} version, the availability of ϵ -T_EX means using a mode test can be done at for example the start of an `\halign`. The `\quitvmode` primitive essentially wraps the same code up at the engine level.

```

23581 \cs_new_protected:Npx \mode_leave_vertical:
23582 {
23583     \cs_if_exist:NTF \pdfTeX_quitvmode:D
23584     { \pdfTeX_quitvmode:D }
23585     {
23586         \exp_not:n
23587         {
23588             \if_mode_vertical:
23589             \exp_after:wN \tex_indent:D
23590             \fi:
23591         }
23592     }
23593 }

```

(End definition for `\mode_leave_vertical:`. This function is documented on page 235.)

43.2 Additions to l3box

```

23594 <@@=box>

```

43.2.1 Viewing part of a box

\box_clip:N A wrapper around the driver-dependent code.

```

\box_clip:c 23595 \cs_new_protected:Npn \box_clip:N #1

```

```

23596 { \hbox_set:Nn #1 { \driver_box_use_clip:N #1 } }
23597 \cs_generate_variant:Nn \box_clip:N { c }

```

(End definition for `\box_clip:N`. This function is documented on page 236.)

`\box_trim:Nnnnn` Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

`\box_trim:cnnnn`

```

23598 \cs_new_protected:Npn \box_trim:Nnnnn #1#2#3#4#5
23599 {
23600   \hbox_set:Nn \l__box_internal_box
23601   {
23602     \tex_kern:D - \__box_dim_eval:n {#2}
23603     \box_use:N #1
23604     \tex_kern:D - \__box_dim_eval:n {#4}
23605   }

```

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim. First, the bottom edge. If there is enough depth, simply set the depth, or if not move down so the result is zero depth. `\box_move_down:nn` is used in both cases so the resulting box always contains a `\lower` primitive. The internal box is used here as it allows safe use of `\box_set_dp:Nn`.

```

23606   \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
23607   {
23608     \hbox_set:Nn \l__box_internal_box
23609     {
23610       \box_move_down:nn \c_zero_dim
23611       { \box_use:N \l__box_internal_box }
23612     }
23613     \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
23614   }
23615   {
23616     \hbox_set:Nn \l__box_internal_box
23617     {
23618       \box_move_down:nn { (#3) - \box_dp:N #1 }
23619       { \box_use:N \l__box_internal_box }
23620     }
23621     \box_set_dp:Nn \l__box_internal_box \c_zero_dim
23622   }

```

Same thing, this time from the top of the box.

```

23623   \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
23624   {
23625     \hbox_set:Nn \l__box_internal_box
23626     {
23627       \box_move_up:nn \c_zero_dim
23628       { \box_use:N \l__box_internal_box }
23629     }
23630     \box_set_ht:Nn \l__box_internal_box
23631     { \box_ht:N \l__box_internal_box - (#5) }
23632   }
23633   {
23634     \hbox_set:Nn \l__box_internal_box
23635     {
23636       \box_move_up:nn { (#5) - \box_ht:N \l__box_internal_box }

```



```

23637         { \box_use:N \l__box_internal_box }
23638     }
23639     \box_set_ht:Nn \l__box_internal_box \c_zero_dim
23640 }
23641 \box_set_eq:NN #1 \l__box_internal_box
23642 }
23643 \cs_generate_variant:Nn \box_trim:Nnnnn { c }

```

(End definition for `\box_trim:Nnnnn`. This function is documented on page 236.)

`\box_viewport:Nnnnn` The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

```

\box_viewport:cnnnn
23644 \cs_new_protected:Npn \box_viewport:Nnnnn #1#2#3#4#5
23645 {
23646     \hbox_set:Nn \l__box_internal_box
23647     {
23648         \tex_kern:D - \__box_dim_eval:n {#2}
23649         \box_use:N #1
23650         \tex_kern:D \__box_dim_eval:n { #4 - \box_wd:N #1 }
23651     }
23652     \dim_compare:nNnTF {#3} < \c_zero_dim
23653     {
23654         \hbox_set:Nn \l__box_internal_box
23655         {
23656             \box_move_down:nn \c_zero_dim
23657             { \box_use:N \l__box_internal_box }
23658         }
23659         \box_set_dp:Nn \l__box_internal_box { - \__box_dim_eval:n {#3} }
23660     }
23661     {
23662         \hbox_set:Nn \l__box_internal_box
23663         { \box_move_down:nn {#3} { \box_use:N \l__box_internal_box } }
23664         \box_set_dp:Nn \l__box_internal_box \c_zero_dim
23665     }
23666     \dim_compare:nNnTF {#5} > \c_zero_dim
23667     {
23668         \hbox_set:Nn \l__box_internal_box
23669         {
23670             \box_move_up:nn \c_zero_dim
23671             { \box_use:N \l__box_internal_box }
23672         }
23673         \box_set_ht:Nn \l__box_internal_box
23674         {
23675             (#5)
23676             \dim_compare:nNnT {#3} > \c_zero_dim
23677             { - (#3) }
23678         }
23679     }
23680     {
23681         \hbox_set:Nn \l__box_internal_box
23682         {
23683             \box_move_up:nn { - \__box_dim_eval:n {#5} }
23684             { \box_use:N \l__box_internal_box }
23685         }

```

```

23686         \box_set_ht:Nn \l__box_internal_box \c_zero_dim
23687     }
23688     \box_set_eq:NN #1 \l__box_internal_box
23689 }
23690 \cs_generate_variant:Nn \box_viewport:Nnnnn { c }

```

(End definition for `\box_viewport:Nnnnn`. This function is documented on page 236.)

43.3 Additions to `l3clist`

```

23691 <@@=clist>

```

`\clist_rand_item:n` The `N`-type function is not implemented through the `n`-type function for efficiency: for instance comma-list variables do not require space-trimming of their items. Even testing for emptiness of an `n`-type comma-list is slow, so we count items first and use that both for the emptiness test and the pseudo-random integer. Importantly, `\clist_item:Nn` and `\clist_item:nn` only evaluate their argument once.

```

23692 \cs_new:Npn \clist_rand_item:n #1
23693 { \exp_args:Nf \__clist_rand_item:nn { \clist_count:n {#1} } {#1} }
23694 \cs_new:Npn \__clist_rand_item:nn #1#2
23695 {
23696     \int_compare:nNf {#1} = 0
23697     { \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }
23698 }
23699 \cs_new:Npn \clist_rand_item:N #1
23700 {
23701     \clist_if_empty:NF #1
23702     { \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }
23703 }
23704 \cs_generate_variant:Nn \clist_rand_item:N { c }

```

(End definition for `\clist_rand_item:n`, `\clist_rand_item:N`, and `__clist_rand_item:nn`. These functions are documented on page 236.)

43.4 Additions to `l3coffins`

```

23705 <@@=coffin>

```

43.4.1 Rotating coffins

`\l__coffin_sin_fp` Used for rotations to get the sine and cosine values.

```

\l__coffin_cos_fp
23706 \fp_new:N \l__coffin_sin_fp
23707 \fp_new:N \l__coffin_cos_fp

```

(End definition for `\l__coffin_sin_fp` and `\l__coffin_cos_fp`.)

`\l__coffin_bounding_prop` A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```

23708 \prop_new:N \l__coffin_bounding_prop

```

(End definition for `\l__coffin_bounding_prop`.)

`\l__coffin_bounding_shift_dim` The shift of the bounding box of a coffin from the real content.

```

23709 \dim_new:N \l__coffin_bounding_shift_dim

```

(End definition for `\l__coffin_bounding_shift_dim`.)

$\backslash l_coffin_left_corner_dim$ $\backslash l_coffin_right_corner_dim$ $\backslash l_coffin_bottom_corner_dim$ $\backslash l_coffin_top_corner_dim$	<p>These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.</p> <pre> 23710 \dim_new:N \l__coffin_left_corner_dim 23711 \dim_new:N \l__coffin_right_corner_dim 23712 \dim_new:N \l__coffin_bottom_corner_dim 23713 \dim_new:N \l__coffin_top_corner_dim </pre>
--	---

(End definition for $\backslash l_coffin_left_corner_dim$ and others.)

$\backslash coffin_rotate:Nn$ $\backslash coffin_rotate:cn$	<p>Rotating a coffin requires several steps which can be conveniently run together. The sine and cosine of the angle in degrees are computed. This is then used to set $\backslash l_coffin_sin_fp$ and $\backslash l_coffin_cos_fp$, which are carried through unchanged for the rest of the procedure.</p>
--	--

```

23714 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
23715 {
23716     \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
23717     \fp_set:Nn \l__coffin_cos_fp { cosd ( #2 ) }

```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```

23718 \prop_map_inline:cn { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
23719 { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
23720 \prop_map_inline:cn { l__coffin_poles_ \__coffin_to_value:N #1 _prop }
23721 { \__coffin_rotate_pole:Nnnnn #1 {##1} ##2 }

```

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

```

23722 \__coffin_set_bounding:N #1
23723 \prop_map_inline:Nn \l__coffin_bounding_prop
23724 { \__coffin_rotate_bounding:nnn {##1} ##2 }

```

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.

```

23725 \__coffin_find_corner_maxima:N #1
23726 \__coffin_find_bounding_shift:
23727 \box_rotate:Nn #1 {#2}

```

The correction of the box position itself takes place here. The idea is that the bounding box for a coffin is tight up to the content, and has the reference point at the bottom-left. The x -direction is handled by moving the content by the difference in the positions of the bounding box and the content left edge. The y -direction is dealt with by moving the box down by any depth it has acquired. The internal box is used here to allow for the next step.

```

23728 \hbox_set:Nn \l__coffin_internal_box
23729 {
23730     \tex_kern:D
23731     \dim_eval:n
23732     { \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim }
23733     \exp_stop_f:
23734     \box_move_down:nn { \l__coffin_bottom_corner_dim }
23735     { \box_use:N #1 }
23736 }

```

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content. As this operation requires setting box dimensions and these transcend grouping, the safe way to do this is to use the internal box and to reset the result into the target box.

```

23737 \box_set_ht:Nn \l__coffin_internal_box
23738 { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
23739 \box_set_dp:Nn \l__coffin_internal_box { 0 pt }
23740 \box_set_wd:Nn \l__coffin_internal_box
23741 { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
23742 \hbox_set:Nn #1 { \box_use:N \l__coffin_internal_box }

```

The final task is to move the poles and corners such that they are back in alignment with the box reference point.

```

23743 \prop_map_inline:cn { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
23744 { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
23745 \prop_map_inline:cn { l__coffin_poles_ \__coffin_to_value:N #1 _prop }
23746 { \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }
23747 }
23748 \cs_generate_variant:Nn \coffin_rotate:Nn { c }

```

(End definition for `\coffin_rotate:Nn`. This function is documented on page 237.)

`__coffin_set_bounding:N` The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

```

23749 \cs_new_protected:Npn \__coffin_set_bounding:N #1
23750 {
23751   \prop_put:Nnx \l__coffin_bounding_prop { tl }
23752   { { 0 pt } { \dim_eval:n { \box_ht:N #1 } } }
23753   \prop_put:Nnx \l__coffin_bounding_prop { tr }
23754   {
23755     { \dim_eval:n { \box_wd:N #1 } }
23756     { \dim_eval:n { \box_ht:N #1 } }
23757   }
23758   \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
23759   \prop_put:Nnx \l__coffin_bounding_prop { bl }
23760   { { 0 pt } { \dim_use:N \l__coffin_internal_dim } }
23761   \prop_put:Nnx \l__coffin_bounding_prop { br }
23762   {
23763     { \dim_eval:n { \box_wd:N #1 } }
23764     { \dim_use:N \l__coffin_internal_dim }
23765   }
23766 }

```

(End definition for `__coffin_set_bounding:N`.)

`__coffin_rotate_bounding:nnn` Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

`__coffin_rotate_corner:Nnnn`

```

23767 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
23768 {
23769   \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
23770   \prop_put:Nnx \l__coffin_bounding_prop {#1}
23771   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }

```

```

23772 }
23773 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
23774 {
23775   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
23776   \prop_put:cnx { \l__coffin_corners_ \__coffin_to_value:N #1 _prop } {#2}
23777   { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
23778 }

```

(End definition for __coffin_rotate_bounding:nnn and __coffin_rotate_corner:Nnnn.)

__coffin_rotate_pole:Nnnnnn Rotating a single pole simply means shifting the co-ordinate of the pole and its direction. The rotation here is about the bottom-left corner of the coffin.

```

23779 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
23780 {
23781   \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
23782   \__coffin_rotate_vector:nnNN {#5} {#6}
23783   \l__coffin_x_prime_dim \l__coffin_y_prime_dim
23784   \__coffin_set_pole:Nnx #1 {#2}
23785   {
23786     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
23787     { \dim_use:N \l__coffin_x_prime_dim }
23788     { \dim_use:N \l__coffin_y_prime_dim }
23789   }
23790 }

```

(End definition for __coffin_rotate_pole:Nnnnnn.)

__coffin_rotate_vector:nnNN A rotation function, which needs only an input vector (as dimensions) and an output space. The values \l__coffin_cos_fp and \l__coffin_sin_fp should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

```

23791 \cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
23792 {
23793   \dim_set:Nn #3
23794   {
23795     \fp_to_dim:n
23796     {
23797       \dim_to_fp:n {#1} * \l__coffin_cos_fp
23798       - \dim_to_fp:n {#2} * \l__coffin_sin_fp
23799     }
23800   }
23801   \dim_set:Nn #4
23802   {
23803     \fp_to_dim:n
23804     {
23805       \dim_to_fp:n {#1} * \l__coffin_sin_fp
23806       + \dim_to_fp:n {#2} * \l__coffin_cos_fp
23807     }
23808   }
23809 }

```

(End definition for __coffin_rotate_vector:nnNN.)

`_coffin_find_corner_maxima:N
 _coffin_find_corner_maxima_aux:nn`

The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```

23810 \cs_new_protected:Npn \_coffin_find_corner_maxima:N #1
23811 {
23812     \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
23813     \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
23814     \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
23815     \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
23816     \prop_map_inline:cn { l__coffin_corners_ \_coffin_to_value:N #1 _prop }
23817     { \_coffin_find_corner_maxima_aux:nn ##2 }
23818 }
23819 \cs_new_protected:Npn \_coffin_find_corner_maxima_aux:nn #1#2
23820 {
23821     \dim_set:Nn \l__coffin_left_corner_dim
23822     { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
23823     \dim_set:Nn \l__coffin_right_corner_dim
23824     { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
23825     \dim_set:Nn \l__coffin_bottom_corner_dim
23826     { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
23827     \dim_set:Nn \l__coffin_top_corner_dim
23828     { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
23829 }
    
```

(End definition for `_coffin_find_corner_maxima:N` and `_coffin_find_corner_maxima_aux:nn`.)

`_coffin_find_bounding_shift:
 _coffin_find_bounding_shift_aux:nn`

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

```

23830 \cs_new_protected:Npn \_coffin_find_bounding_shift:
23831 {
23832     \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
23833     \prop_map_inline:Nn \l__coffin_bounding_prop
23834     { \_coffin_find_bounding_shift_aux:nn ##2 }
23835 }
23836 \cs_new_protected:Npn \_coffin_find_bounding_shift_aux:nn #1#2
23837 {
23838     \dim_set:Nn \l__coffin_bounding_shift_dim
23839     { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
23840 }
    
```

(End definition for `_coffin_find_bounding_shift:` and `_coffin_find_bounding_shift_aux:nn`.)

`_coffin_shift_corner:Nnnn
 _coffin_shift_pole:Nnnnnn`

Shifting the corners and poles of a coffin means subtracting the appropriate values from the x - and y -components. For the poles, this means that the direction vector is unchanged.

```

23841 \cs_new_protected:Npn \_coffin_shift_corner:Nnnn #1#2#3#4
23842 {
23843     \prop_put:cnx { l__coffin_corners_ \_coffin_to_value:N #1 _prop } {#2}
23844     {
23845         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
23846         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
23847     }
    
```

```

23848 }
23849 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
23850 {
23851   \prop_put:cnx { l__coffin_poles_ \__coffin_to_value:N #1 _ prop } {#2}
23852   {
23853     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
23854     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
23855     {#5} {#6}
23856   }
23857 }

```

(End definition for __coffin_shift_corner:Nnnn and __coffin_shift_pole:Nnnnnn.)

43.4.2 Resizing coffins

\l__coffin_scale_x_fp Storage for the scaling factors in x and y , respectively.
 \l__coffin_scale_y_fp

```

23858 \fp_new:N \l__coffin_scale_x_fp
23859 \fp_new:N \l__coffin_scale_y_fp

```

(End definition for \l__coffin_scale_x_fp and \l__coffin_scale_y_fp.)

\l__coffin_scaled_total_height_dim When scaling, the values given have to be turned into absolute values.
 \l__coffin_scaled_width_dim

```

23860 \dim_new:N \l__coffin_scaled_total_height_dim
23861 \dim_new:N \l__coffin_scaled_width_dim

```

(End definition for \l__coffin_scaled_total_height_dim and \l__coffin_scaled_width_dim.)

\coffin_resize:Nnn Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.
\coffin_resize:cnn

```

23862 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
23863 {
23864   \fp_set:Nn \l__coffin_scale_x_fp
23865   { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
23866   \fp_set:Nn \l__coffin_scale_y_fp
23867   {
23868     \dim_to_fp:n {#3}
23869     / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
23870   }
23871   \box_resize_to_wd_and_ht_plus_dp:Nnn #1 {#2} {#3}
23872   \__coffin_resize_common:Nnn #1 {#2} {#3}
23873 }
23874 \cs_generate_variant:Nn \coffin_resize:Nnn { c }

```

(End definition for \coffin_resize:Nnn. This function is documented on page [237](#).)

__coffin_resize_common:Nnn The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```

23875 \cs_new_protected:Npn \__coffin_resize_common:Nnn #1#2#3
23876 {
23877   \prop_map_inline:cn { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
23878   { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
23879   \prop_map_inline:cn { l__coffin_poles_ \__coffin_to_value:N #1 _prop }
23880   { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }

```

Negative x -scaling values place the poles in the wrong location: this is corrected here.

```

23881 \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
23882 {
23883   \prop_map_inline:cn
23884     { l__coffin_corners_ \__coffin_to_value:N #1 _prop }
23885     { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
23886   \prop_map_inline:cn
23887     { l__coffin_poles_ \__coffin_to_value:N #1 _prop }
23888     { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
23889 }
23890 }

```

(End definition for `__coffin_resize_common:Nnn`.)

`\coffin_scale:Nnn`
`\coffin_scale:cnn`

For scaling, the opposite calculation is done to find the new dimensions for the coffin. Only the total height is needed, as this is the shift required for corners and poles. The scaling is done the T_EX way as this works properly with floating point values without needing to use the `fp` module.

```

23891 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
23892 {
23893   \fp_set:Nn \l__coffin_scale_x_fp {#2}
23894   \fp_set:Nn \l__coffin_scale_y_fp {#3}
23895   \box_scale:Nnn #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
23896   \dim_set:Nn \l__coffin_internal_dim
23897     { \coffin_ht:N #1 + \coffin_dp:N #1 }
23898   \dim_set:Nn \l__coffin_scaled_total_height_dim
23899     { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
23900   \dim_set:Nn \l__coffin_scaled_width_dim
23901     { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
23902   \__coffin_resize_common:Nnn #1
23903     { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
23904 }
23905 \cs_generate_variant:Nn \coffin_scale:Nnn { c }

```

(End definition for `\coffin_scale:Nnn`. This function is documented on page 237.)

`__coffin_scale_vector:nnNN`

This functions scales a vector from the origin using the pre-set scale factors in x and y . This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```

23906 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
23907 {
23908   \dim_set:Nn #3
23909     { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
23910   \dim_set:Nn #4
23911     { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
23912 }

```

(End definition for `__coffin_scale_vector:nnNN`.)

`__coffin_scale_corner:Nnnn`
`__coffin_scale_pole:Nnnnnn`

Scaling both corners and poles is a simple calculation using the preceding vector scaling.

```

23913 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
23914 {
23915   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
23916   \prop_put:cnx { l__coffin_corners_ \__coffin_to_value:N #1 _prop } {#2}

```



```

23917     { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
23918   }
23919 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
23920 {
23921   \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
23922   \__coffin_set_pole:Nnx #1 {#2}
23923   {
23924     { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
23925     {#5} {#6}
23926   }
23927 }

```

(End definition for `__coffin_scale_corner:Nnnn` and `__coffin_scale_pole:Nnnnnn`.)

```

\__coffin_x_shift_corner:Nnnn
\__coffin_x_shift_pole:Nnnnnn

```

These functions correct for the x displacement that takes place with a negative horizontal scaling.

```

23928 \cs_new_protected:Npn \__coffin_x_shift_corner:Nnnn #1#2#3#4
23929 {
23930   \prop_put:cnx { l__coffin_corners_ \__coffin_to_value:N #1 _prop } {#2}
23931   {
23932     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
23933   }
23934 }
23935 \cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
23936 {
23937   \prop_put:cnx { l__coffin_poles_ \__coffin_to_value:N #1 _prop } {#2}
23938   {
23939     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
23940     {#5} {#6}
23941   }
23942 }

```

(End definition for `__coffin_x_shift_corner:Nnnn` and `__coffin_x_shift_pole:Nnnnnn`.)

43.5 Additions to `l3file`

```

23943 <@@=file>

```

```

\file_get_md5five_hash:nN
\file_get_size:nN
\file_get_timestamp:nN
\__file_get_details:nnN

```

These are all wrappers around the pdfTeX primitives doing the same jobs: as we want consistent file paths to be found, they are all set up using `\file_get_full_name:nN` and so are non-expandable `get` functions. Much of the code is repetitive but we need to branch for LuaTeX (emulation in Lua), for the slightly different syntax needed for `\pdfTeX_md5fivesum:D` and for the fact that primitive coverage varies in other engines.

```

23944 \cs_new_protected:Npn \file_get_md5five_hash:nN #1#2
23945 { \__file_get_details:nnN {#1} { md5fivesum } {#2} }
23946 \cs_new_protected:Npn \file_get_size:nN #1#2
23947 { \__file_get_details:nnN {#1} { size } {#2} }
23948 \cs_new_protected:Npn \file_get_timestamp:nN #1#2
23949 { \__file_get_details:nnN {#1} { moddate } {#2} }
23950 \cs_new_protected:Npn \__file_get_details:nnN #1#2#3
23951 {
23952   \file_get_full_name:nN {#1} \l__file_full_name_str
23953   \str_set:Nx #3
23954   {

```

```

23955         \use:c { pdftex_file #2 :D } \exp_after:wN
23956         { \l__file_full_name_str }
23957     }
23958 }
23959 \sys_if_engine luatex:TF
23960 {
23961     \cs_set_protected:Npn \__file_get_details:nnN #1#2#3
23962     {
23963         \file_get_full_name:nN {#1} \l__file_full_name_str
23964         \str_set:Nx #3
23965         {
23966             \lua_now_x:n
23967             {
23968                 l3kernel.file#2
23969                 ( " \lua_escape_x:n { \l__file_full_name_str } " )
23970             }
23971         }
23972     }
23973 }
23974 {
23975     \cs_set_protected:Npn \file_get_md5hash:nN #1#2
23976     {
23977         \file_get_full_name:nN {#1} \l__file_full_name_str
23978         \tl_set:Nx #2
23979         {
23980             \pdftex_md5sum:D file \exp_after:wN
23981             { \l__file_full_name_str }
23982         }
23983     }
23984     \sys_if_engine xetex:T
23985     {
23986         \cs_set_protected:Npn \__file_get_details:nnN #1#2#3
23987         {
23988             \tl_clear:N #3
23989             \__kernel_msg_error:nnx
23990             { kernel } { xetex-primitive-not-available }
23991             { \exp_not:c { pdf_file #2 } }
23992         }
23993     }
23994 }
23995 \__kernel_msg_new:nnnn { kernel } { xetex-primitive-not-available }
23996 { Primitive~\token_to_str:N #1 not-available }
23997 {
23998     XeTeX~does~not~currently~provide~functionality~equivalent~to~the~
23999     \token_to_str:N #1 primitive.
24000 }

```

(End definition for \file_get_md5hash:nN and others. These functions are documented on page 237.)

\file_if_exist_input:n Input of a file with a test for existence. We do not define the T or TF variants because the most useful place to place the *<true code>* would be inconsistent with other conditionals.

```

24001 \cs_new_protected:Npn \file_if_exist_input:n #1
24002 {

```

```

24003 \file_get_full_name:nN {#1} \l__file_full_name_str
24004 \str_if_empty:NF \l__file_full_name_str
24005 { \__file_input:V \l__file_full_name_str }
24006 }
24007 \cs_new_protected:Npn \file_if_exist_input:nF #1#2
24008 {
24009 \file_get_full_name:nN {#1} \l__file_full_name_str
24010 \str_if_empty:NTF \l__file_full_name_str
24011 {#2}
24012 { \__file_input:V \l__file_full_name_str }
24013 }

```

(End definition for `\file_if_exist_input:n` and `\file_if_exist_input:nF`. These functions are documented on page 238.)

`\file_input_stop:` A simple rename.

```

24014 \cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }

```

(End definition for `\file_input_stop:`. This function is documented on page 238.)

43.6 Additions to `l3flag`

```

24015 <@@=flag>

```

`\flag_raise_if_clear:n` It might be faster to just call the “trap” function in all cases but conceptually the function name suggests we should only run it if the flag is zero in case the “trap” made customizable in the future.

```

24016 \__kernel_patch:nnNNpn { \__flag_chk_exist:n {#1} } { }
24017 \cs_new:Npn \flag_raise_if_clear:n #1
24018 {
24019 \if_cs_exist:w flag~#1~0 \cs_end:
24020 \else:
24021 \cs:w flag~#1 \cs_end: 0 ;
24022 \fi:
24023 }

```

(End definition for `\flag_raise_if_clear:n`. This function is documented on page 238.)

43.7 Additions to `l3msg`

```

24024 <@@=msg>

```

Pass to an auxiliary the message to display and the module name

```

\msg_expandable_error:nnnnnn
\msg_expandable_error:nnffff
\msg_expandable_error:nnnnn
\msg_expandable_error:nnfff
\msg_expandable_error:nnnn
\msg_expandable_error:nnff
\msg_expandable_error:nnn
\msg_expandable_error:nnf
\msg_expandable_error:nn
\__msg_expandable_error_module:nn
24025 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
24026 {
24027 \exp_args:Nf \__msg_expandable_error_module:nn
24028 {
24029 \exp_args:Nf \tl_to_str:n
24030 { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
24031 }
24032 {#1}
24033 }
24034 \cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
24035 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
24036 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4

```

```

24037 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
24038 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
24039 { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
24040 \cs_new:Npn \msg_expandable_error:nn #1#2
24041 { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
24042 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
24043 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnfff }
24044 \cs_generate_variant:Nn \msg_expandable_error:nnnn { nnff }
24045 \cs_generate_variant:Nn \msg_expandable_error:nnnn { nnf }
24046 \cs_new:Npn \__msg_expandable_error_module:nn #1#2
24047 {
24048   \exp_after:wN \exp_after:wN
24049   \exp_after:wN \use_none_delimit_by_q_stop:w
24050   \use:n { \::error ! ~ #2 : ~ #1 } \q_stop
24051 }

```

(End definition for `\msg_expandable_error:nnnnnn` and others. These functions are documented on page 239.)

`\msg_show_eval:Nn` A short-hand used for `\int_show:n` and similar functions that passes to `\tl_show:n` the result of applying #1 (a function such as `\int_eval:n`) to the expression #2. The use of `f`-expansion ensures that #1 is expanded in the scope in which the show command is called, rather than in the group created by `\iow_wrap:nnnN`. This is only important for expressions involving the `\currentgrouplevel` or `\currentgrouptype`. On the other hand we want the expression to be converted to a string with the usual escape character, hence within the wrapping code.

```

24052 \cs_new_protected:Npn \msg_show_eval:Nn #1#2
24053 { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_show:n }
24054 \cs_new_protected:Npn \msg_log_eval:Nn #1#2
24055 { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_log:n }
24056 \cs_new_protected:Npn \__msg_show_eval:nnN #1#2#3 { #3 { #2 = #1 } }

```

(End definition for `\msg_show_eval:Nn`, `\msg_log_eval:Nn`, and `__msg_show_eval:nnN`. These functions are documented on page 239.)

`\msg_show_item:n` Each item in the variable is formatted using one of the following functions. We cannot use `\` and so on because these short-hands cannot be used inside the arguments of messages, only when defining the messages.

```

\msg_show_item_unbraced:n
\msg_show_item:nn
\msg_show_item_unbraced:nn
24057 \cs_new:Npx \msg_show_item:n #1
24058 { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n { {#1} } }
24059 \cs_new:Npx \msg_show_item_unbraced:n #1
24060 { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n {#1} }
24061 \cs_new:Npx \msg_show_item:nn #1#2
24062 {
24063   \iow_newline: > \use:nn { ~ } { ~ }
24064   \exp_not:N \tl_to_str:n { {#1} }
24065   \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
24066   \exp_not:N \tl_to_str:n { {#2} }
24067 }
24068 \cs_new:Npx \msg_show_item_unbraced:nn #1#2
24069 {
24070   \iow_newline: > \use:nn { ~ } { ~ }
24071   \exp_not:N \tl_to_str:n {#1}
24072   \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }

```

```

24073 \exp_not:N \tl_to_str:n {#2}
24074 }

```

(End definition for `\msg_show_item:n` and others. These functions are documented on page 239.)

43.8 Additions to l3prg

\bool_const:Nn A merger between `\tl_const:Nn` and `\bool_set:Nn`.
\bool_const:cn

```

24075 \__kernel_patch:nnNnpn { \__kernel_chk_var_scope:NN c #1 } { }
24076 \cs_new_protected:Npn \bool_const:Nn #1#2
24077 {
24078   \__kernel_chk_if_free_cs:N #1
24079   \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
24080 }
24081 \cs_generate_variant:Nn \bool_const:Nn { c }

```

(End definition for `\bool_const:Nn`. This function is documented on page 240.)

43.9 Additions to l3prop

```

24082 <@@=prop>

```

\prop_count:N Counting the key–value pairs in a property list is done using the same approach as for
\prop_count:c other count functions: turn each entry into a +1 then use integer evaluation to actually
__prop_count:nn do the mathematics.

```

24083 \cs_new:Npn \prop_count:N #1
24084 {
24085   \int_eval:n
24086   {
24087     0
24088     \prop_map_function:NN #1 \__prop_count:nn
24089   }
24090 }
24091 \cs_new:Npn \__prop_count:nn #1#2 { + 1 }
24092 \cs_generate_variant:Nn \prop_count:N { c }

```

(End definition for `\prop_count:N` and `__prop_count:nn`. This function is documented on page 240.)

\prop_map_tokens:Nn The mapping is very similar to `\prop_map_function:NN`. The `\use_i:nn` removes the
\prop_map_tokens:cn leading `\s__prop`. The odd construction `\use:n {#1}` allows #1 to contain any token
__prop_map_tokens:nwn without interfering with `\prop_map_break:.` The loop stops when the argument delimited by `__prop_pair:wn` is `\prg_break:` instead of being empty.

```

24093 \cs_new:Npn \prop_map_tokens:Nn #1#2
24094 {
24095   \exp_last_unbraced:Nno
24096   \use_i:nn { \__prop_map_tokens:nwn {#2} } #1
24097   \prg_break: \__prop_pair:wn \s__prop { } \prg_break_point:
24098   \prg_break_point:Nn \prop_map_break: { }
24099 }
24100 \cs_new:Npn \__prop_map_tokens:nwn #1#2 \__prop_pair:wn #3 \s__prop #4
24101 {
24102   #2
24103   \use:n {#1} {#3} {#4}
24104   \__prop_map_tokens:nwn {#1}

```

```

24105 }
24106 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }

```

(End definition for `\prop_map_tokens:Nn` and `__prop_map_tokens:nwn`. This function is documented on page 240.)

```

\prop_rand_key_value:N
\prop_rand_key_value:c
  \__prop_rand_item:w

```

Contrarily to `clist`, `seq` and `tl`, there is no function to get an item of a `prop` given an integer between 1 and the number of items, so we write the appropriate code. There is no bounds checking because `\int_rand:nn` is always within bounds. The initial `\int_value:w` is stopped by the first `\s__prop` in #1.

```

24107 \cs_new:Npn \prop_rand_key_value:N #1
24108 {
24109   \prop_if_empty:NF #1
24110   {
24111     \exp_after:wN \__prop_rand_item:w
24112     \int_value:w \int_rand:nn { 1 } { \prop_count:N #1 }
24113     #1 \q_stop
24114   }
24115 }
24116 \cs_generate_variant:Nn \prop_rand_key_value:N { c }
24117 \cs_new:Npn \__prop_rand_item:w #1 \s__prop \__prop_pair:wn #2 \s__prop #3
24118 {
24119   \int_compare:nNnF {#1} > 1
24120   { \use_i_delimit_by_q_stop:nw { \exp_not:n { {#2} {#3} } } }
24121   \exp_after:wN \__prop_rand_item:w
24122   \int_value:w \int_eval:n { #1 - 1 } \s__prop
24123 }

```

(End definition for `\prop_rand_key_value:N` and `__prop_rand_item:w`. This function is documented on page 240.)

43.10 Additions to `l3seq`

```

24124 <@@=seq>

```

```

\seq_mapthread_function:NNN
\seq_mapthread_function:NcN
\seq_mapthread_function:cNN
\seq_mapthread_function:ccN
  \__seq_mapthread_function:wNN
  \__seq_mapthread_function:wNw
  \__seq_mapthread_function:Nnnwnn

```

The idea is to first expand both sequences, adding the usual `{ ? \prg_break: } { }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of #2 and #5, which for items in both sequences are `\s__seq` `__seq_item:n`. The function to be mapped are then be applied to the two entries. When the code hits the end of one of the sequences, the break material stops the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```

24125 \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
24126 { \exp_after:wN \__seq_mapthread_function:wNN #2 \q_stop #1 #3 }
24127 \cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \q_stop #2#3
24128 {
24129   \exp_after:wN \__seq_mapthread_function:wNw #2 \q_stop #3
24130   #1 { ? \prg_break: } { }
24131   \prg_break_point:
24132 }
24133 \cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \q_stop #2
24134 {
24135   \__seq_mapthread_function:Nnnwnn #2
24136   #1 { ? \prg_break: } { }

```

```

24137     \q_stop
24138   }
24139 \cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \q_stop #5#6
24140 {
24141     \use_none:n #2
24142     \use_none:n #5
24143     #1 {#3} {#6}
24144     \__seq_mapthread_function:Nnnwnn #1 #4 \q_stop
24145 }
24146 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc , c , cc }

```

(End definition for `\seq_mapthread_function:NNN` and others. This function is documented on page 241.)

`\seq_set_filter:NNn` Similar to `\seq_map_inline:Nn`, without a `\prg_break_point:` because the user's code is performed within the evaluation of a boolean expression, and skipping out of that would break horribly. The `__seq_wrap_item:n` function inserts the relevant `__seq_item:n` without expansion in the input stream, hence in the x-expanding assignment.

`\seq_gset_filter:NNn`

`__seq_set_filter:NNNn`

```

24147 \cs_new_protected:Npn \seq_set_filter:NNn
24148 { \__seq_set_filter:NNNn \tl_set:Nx }
24149 \cs_new_protected:Npn \seq_gset_filter:NNn
24150 { \__seq_set_filter:NNNn \tl_gset:Nx }
24151 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
24152 {
24153     \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
24154     #1 #2 { #3 }
24155     \__seq_pop_item_def:
24156 }

```

(End definition for `\seq_set_filter:NNn`, `\seq_gset_filter:NNn`, and `__seq_set_filter:NNNn`. These functions are documented on page 241.)

`\seq_set_map:NNn` Very similar to `\seq_set_filter:NNn`. We could actually merge the two within a single function, but it would have weird semantics.

`\seq_gset_map:NNn`

`__seq_set_map:NNNn`

```

24157 \cs_new_protected:Npn \seq_set_map:NNn
24158 { \__seq_set_map:NNNn \tl_set:Nx }
24159 \cs_new_protected:Npn \seq_gset_map:NNn
24160 { \__seq_set_map:NNNn \tl_gset:Nx }
24161 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
24162 {
24163     \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
24164     #1 #2 { #3 }
24165     \__seq_pop_item_def:
24166 }

```

(End definition for `\seq_set_map:NNn`, `\seq_gset_map:NNn`, and `__seq_set_map:NNNn`. These functions are documented on page 241.)

`\seq_set_from_inline_x:Nnn` Set `__seq_item:n` then map it using the loop code.

`\seq_gset_from_inline_x:Nnn`

`__seq_set_from_inline_x:NNnn`

```

24167 \cs_new_protected:Npn \seq_set_from_inline_x:Nnn
24168 { \__seq_set_from_inline_x:NNnn \tl_set:Nx }
24169 \cs_new_protected:Npn \seq_gset_from_inline_x:Nnn
24170 { \__seq_set_from_inline_x:NNnn \tl_gset:Nx }
24171 \cs_new_protected:Npn \__seq_set_from_inline_x:NNnn #1#2#3#4
24172 {

```

```

24173     \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
24174     #1 #2 { \s__seq #3 \__seq_item:n }
24175     \__seq_pop_item_def:
24176 }

```

(End definition for `\seq_set_from_inline_x:Nnn`, `\seq_gset_from_inline_x:Nnn`, and `__seq_set_from_inline_x:NNnn`. These functions are documented on page 242.)

`\seq_set_from_function:NnN`
`\seq_gset_from_function:NnN`

Reuse `\seq_set_from_inline_x:Nnn`.

```

24177 \cs_new_protected:Npn \seq_set_from_function:NnN #1#2#3
24178 { \seq_set_from_inline_x:Nnn #1 {#2} { #3 {##1} } }
24179 \cs_new_protected:Npn \seq_gset_from_function:NnN #1#2#3
24180 { \seq_gset_from_inline_x:Nnn #1 {#2} { #3 {##1} } }

```

(End definition for `\seq_set_from_function:NnN` and `\seq_gset_from_function:NnN`. These functions are documented on page 242.)

`\seq_rand_item:N`
`\seq_rand_item:c`

Importantly, `\seq_item:Nn` only evaluates its argument once.

```

24181 \cs_new:Npn \seq_rand_item:N #1
24182 {
24183     \seq_if_empty:NF #1
24184     { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }
24185 }
24186 \cs_generate_variant:Nn \seq_rand_item:N { c }

```

(End definition for `\seq_rand_item:N`. This function is documented on page 241.)

`\seq_const_from_clist:Nn`
`\seq_const_from_clist:cn`

Almost identical to `\seq_set_from_clist:Nn`.

```

24187 \cs_new_protected:Npn \seq_const_from_clist:Nn #1#2
24188 {
24189     \tl_const:Nx #1
24190     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
24191 }
24192 \cs_generate_variant:Nn \seq_const_from_clist:Nn { c }

```

(End definition for `\seq_const_from_clist:Nn`. This function is documented on page 241.)

`\seq_shuffle:N`
`\seq_gshuffle:N`

We apply the Fisher–Yates shuffle, storing items in `\toks` registers. We use the primitive `\pdfTeX_uniformdeviate:D` for speed reasons. Its non-uniformity is of order its argument divided by 2^{28} , not too bad for small lists. For sequences with more than 13 elements there are more possible permutations than possible seeds ($13! > 2^{28}$) so the question of uniformity is somewhat moot.

`__seq_shuffle:NN`
`__seq_shuffle_item:n`
`\g__seq_internal_seq`
`\l__seq_internal_a_int`
`\l__seq_internal_b_int`

```

24193 \cs_if_exist:NTF \pdfTeX_uniformdeviate:D
24194 {
24195     \int_new:N \l__seq_internal_a_int
24196     \int_new:N \l__seq_internal_b_int
24197     \seq_new:N \g__seq_internal_seq
24198     \cs_new_protected:Npn \seq_shuffle:N { \__seq_shuffle:NN \seq_set_eq:NN }
24199     \cs_new_protected:Npn \seq_gshuffle:N { \__seq_shuffle:NN \seq_gset_eq:NN }
24200     \cs_new_protected:Npn \__seq_shuffle:NN #1#2
24201     {
24202         \int_compare:nNnTF { \seq_count:N #2 } > \c_max_register_int
24203         {
24204             \__kernel_msg_error:nxx { kernel } { shuffle-too-large }
24205             { \token_to_str:N #2 }

```



```

24206     }
24207     {
24208         \group_begin:
24209             \cs_set_eq:NN \__seq_item:n \__seq_shuffle_item:n
24210             \int_zero:N \l__seq_internal_a_int
24211             #2
24212             \seq_gset_from_inline_x:Nnn \g__seq_internal_seq
24213                 { \int_step_function:nN { \l__seq_internal_a_int } }
24214                 { \tex_the:D \tex_toks:D ##1 }
24215         \group_end:
24216         #1 #2 \g__seq_internal_seq
24217         \seq_gclear:N \g__seq_internal_seq
24218     }
24219 }
24220 \cs_new_protected:Npn \__seq_shuffle_item:n
24221 {
24222     \int_incr:N \l__seq_internal_a_int
24223     \int_set:Nn \l__seq_internal_b_int
24224         { 1 + \pdfTeX_uniformdeviate:D \l__seq_internal_a_int }
24225     \tex_toks:D \l__seq_internal_a_int
24226         = \tex_toks:D \l__seq_internal_b_int
24227     \tex_toks:D \l__seq_internal_b_int
24228 }
24229 \__kernel_msg_new:nnnn { kernel } { shuffle-too-large }
24230 { The~sequence~#1~is~too~long~to~be~shuffled~by~TeX. }
24231 {
24232     TeX~has~ \int_eval:n { \c_max_register_int + 1 } ~
24233     toks~registers:~this~only~allows~to~shuffle~up~to~
24234     \int_use:N \c_max_register_int \ items.~
24235     The~list~will~not~be~shuffled.
24236 }
24237 }
24238 {
24239     \cs_new_protected:Npn \seq_shuffle:N #1
24240         { \__kernel_msg_error:nn { kernel } { fp-no-random } }
24241     \cs_new_eq:NN \seq_gshuffle:N \seq_shuffle:N
24242 }

```

(End definition for `\seq_shuffle:N` and others. These functions are documented on page 242.)

43.11 Additions to `l3skip`

```

24243 <@@=skip>

```

`\skip_split_finite_else_action:nnNN` This macro is useful when performing error checking in certain circumstances. If the `<skip>` register holds finite glue it sets #3 and #4 to the stretch and shrink component, resp. If it holds infinite glue set #3 and #4 to zero and issue the special action #2 which is probably an error message. Assignments are local.

```

24244 \cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
24245 {
24246     \skip_if_finite:nTF {#1}
24247     {
24248         #3 = \etex_gluestretch:D #1 \scan_stop:
24249         #4 = \etex_glueshrink:D #1 \scan_stop:

```

```

24250     }
24251     {
24252         #3 = \c_zero_skip
24253         #4 = \c_zero_skip
24254         #2
24255     }
24256 }

```

(End definition for `\skip_split_finite_else_action:n`. This function is documented on page 242.)

43.12 Additions to `l3sys`

```

24257 <@@=sys>

```

`\sys_if_rand_exist:p`: Currently, randomness exists under pdfTeX, LuaTeX, pTeX and upTeX.

```

\sys_if_rand_exist:TF
24258 \__sys_const:nn { sys_if_rand_exist }
24259 { \cs_if_exist:N \pdfTeX_uniformdeviate:D }

```

(End definition for `\sys_if_rand_exist:TF`. This function is documented on page 243.)

`\sys_rand_seed`: Unpack the primitive.

```

24260 \cs_new:Npn \sys_rand_seed: { \tex_the:D \pdfTeX_randomseed:D }
24261 \cs_if_exist:NF \pdfTeX_randomseed:D
24262 { \cs_set:Npn \sys_rand_seed: { 0 } }

```

(End definition for `\sys_rand_seed:`. This function is documented on page 243.)

`\sys_gset_rand_seed:n`: The primitive always assigns the seed globally.

```

24263 \cs_new_protected:Npn \sys_gset_rand_seed:n #1
24264 { \pdfTeX_setrandomseed:D \int_eval:n {#1} \exp_stop_f: }

```

(End definition for `\sys_gset_rand_seed:n`. This function is documented on page 243.)

`\c_sys_shell_escape_int`: Expose the engine’s shell escape status to the user.

```

24265 \int_const:Nn \c_sys_shell_escape_int
24266 {
24267     \sys_if_engine luatex:TF
24268     {
24269         \luaTeX_directlua:D
24270         { tex.sprint(status.shell_escape~or~os.execute()) }
24271     }
24272     {
24273         \pdfTeX_shellescape:D
24274     }
24275 }

```

(End definition for `\c_sys_shell_escape_int`. This variable is documented on page 243.)

`\sys_if_shell_p`: Performs a check for whether shell escape is enabled. The first set of functions returns true if either of restricted or unrestricted shell escape is enabled, while the other two sets of functions return true in only one of these two cases.

```

\sys_if_shell_unrestricted:p
\sys_if_shell_unrestricted:TF
\sys_if_shell_restricted:p
\sys_if_shell_restricted:TF
24276 \__sys_const:nn { sys_if_shell }
24277 { \int_compare_p:nNn \c_sys_shell_escape_int > 0 }
24278 \__sys_const:nn { sys_if_shell_unrestricted }
24279 { \int_compare_p:nNn \c_sys_shell_escape_int = 1 }
24280 \__sys_const:nn { sys_if_shell_restricted }
24281 { \int_compare_p:nNn \c_sys_shell_escape_int = 2 }

```

(End definition for `\sys_if_shell:TF`, `\sys_if_shell_unrestricted:TF`, and `\sys_if_shell_restricted:TF`. These functions are documented on page 243.)

`\c__sys_shell_stream_int` This is not needed for LuaTeX: shell escape there isn't done using a TeX interface

```
24282 \sys_if_engine luatex:F
24283   { \int_const:Nn \c__sys_shell_stream_int { 18 } }
```

(End definition for `\c__sys_shell_stream_int`.)

`\sys_shell_now:n` Execute commands through shell escape immediately.

```
24284 \sys_if_engine luatex:TF
24285   {
24286     \cs_new_protected:Npn \sys_shell_now:n #1
24287     {
24288       \lua_now_x:n
24289       { os.execute(" \lua_escape_x:n { \tl_to_str:n {#1} } ") }
24290     }
24291   }
24292   {
24293     \cs_new_protected:Npn \sys_shell_now:n #1
24294     { \iow_now:Nn \c__sys_shell_stream_int {#1} }
24295   }
24296   \cs_generate_variant:Nn \sys_shell_now:n { x }
```

(End definition for `\sys_shell_now:n`. This function is documented on page 244.)

`\sys_shell_shipout:n` Execute commands through shell escape at shipout.

```
24297 \sys_if_engine luatex:TF
24298   {
24299     \cs_new_protected:Npn \sys_shell_shipout:n #1
24300     {
24301       \lua_shipout_x:n
24302       { os.execute(" \lua_escape_x:n { \tl_to_str:n {#1} } ") }
24303     }
24304   }
24305   {
24306     \cs_new_protected:Npn \sys_shell_shipout:n #1
24307     { \iow_shipout:Nn \c__sys_shell_stream_int {#1} }
24308   }
24309   \cs_generate_variant:Nn \sys_shell_shipout:n { x }
```

(End definition for `\sys_shell_shipout:n`. This function is documented on page 244.)

43.13 Additions to `l3tl`

```
24310 <@@=tl>
```

`\tl_if_single_token_p:n` There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. For the next case, an empty token list is not a single token. Finally, we have a non-empty token list starting with a space or a brace group. Applying f-expansion yields an empty result if and only if the token list is a single space.

`\tl_if_single_token:nTF`

```
24311 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
24312   {
24313     \tl_if_head_is_N_type:nTF {#1}
```

```

24314 { \_tl_if_empty_return:o { \use_none:n #1 } }
24315 {
24316   \tl_if_empty:nTF {#1}
24317   { \prg_return_false: }
24318   { \_tl_if_empty_return:o { \exp:w \exp_end_continue_f:w #1 } }
24319 }
24320 }

```

(End definition for `\tl_if_single_token:nTF`. This function is documented on page 244.)

`\tl_reverse_tokens:n` The same as `\tl_reverse:n` but with recursion within brace groups.

```

\__tl_reverse_group:nn
24321 \cs_new:Npn \tl_reverse_tokens:n #1
24322 {
24323   \__kernel_exp_not:w \exp_after:wN
24324   {
24325     \exp:w
24326     \__tl_act:NNNnn
24327     \__tl_reverse_normal:nN
24328     \__tl_reverse_group:nn
24329     \__tl_reverse_space:n
24330     { }
24331     {#1}
24332   }
24333 }
24334 \cs_new:Npn \__tl_reverse_group:nn #1
24335 {
24336   \__tl_act_group_recurse:Nnn
24337   \__tl_act_reverse_output:n
24338   { \tl_reverse_tokens:n }
24339 }

```

`__tl_act_group_recurse:Nnn` In many applications of `__tl_act:NNNnn`, we need to recursively apply some transformation within brace groups, then output. In this code, `#1` is the output function, `#2` is the transformation, which should expand in two steps, and `#3` is the group.

```

24340 \cs_new:Npn \__tl_act_group_recurse:Nnn #1#2#3
24341 {
24342   \exp_args:Nf #1
24343   { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
24344 }

```

(End definition for `\tl_reverse_tokens:n`, `__tl_reverse_group:nn`, and `__tl_act_group_recurse:Nnn`. This function is documented on page 244.)

`\tl_count_tokens:n` The token count is computed through an `\int_eval:n` construction. Each `1+` is output to the *left*, into the integer expression, and the sum is ended by the `\exp_end:` inserted by `__tl_act_end:wN` (which is technically implemented as `\c_zero`). Somewhat a hack!

```

\__tl_act_count_normal:nN
\__tl_act_count_group:nn
\__tl_act_count_space:n
24345 \cs_new:Npn \tl_count_tokens:n #1
24346 {
24347   \int_eval:n
24348   {
24349     \__tl_act:NNNnn
24350     \__tl_act_count_normal:nN
24351     \__tl_act_count_group:nn
24352     \__tl_act_count_space:n

```

```

24353         { }
24354         {#1}
24355     }
24356 }
24357 \cs_new:Npn \__tl_act_count_normal:nN #1 #2 { 1 + }
24358 \cs_new:Npn \__tl_act_count_space:n #1 { 1 + }
24359 \cs_new:Npn \__tl_act_count_group:nn #1 #2
24360 { 2 + \tl_count_tokens:n {#2} + }

```

(End definition for `\tl_count_tokens:n` and others. This function is documented on page 244.)

`\tl_set_from_file:Nnn` The approach here is similar to that for doing a rescan, and so the same internals can be reused. Thus the plan is to insert a pair of tokens of the same charcode but different catcodes after the file has been read. This plus `\exp_not:N` allows the primitive to be used to carry out a set operation.

```

\__tl_set_from_file:NNnn
  \__tl_from_file_do:w
24361 \cs_new_protected:Npn \tl_set_from_file:Nnn
24362 { \__tl_set_from_file:NNnn \tl_set:Nn }
24363 \cs_new_protected:Npn \tl_gset_from_file:Nnn
24364 { \__tl_set_from_file:NNnn \tl_gset:Nn }
24365 \cs_generate_variant:Nn \tl_set_from_file:Nnn { c }
24366 \cs_generate_variant:Nn \tl_gset_from_file:Nnn { c }
24367 \cs_new_protected:Npn \__tl_set_from_file:NNnn #1#2#3#4
24368 {
24369   \file_get_full_name:nN {#4} \l__tl_file_name_str
24370   \str_if_empty:NTF \l__tl_file_name_str
24371   { \__kernel_file_missing:n {#4} }
24372   {
24373     \group_begin:
24374     \exp_args:No \etex_everyeof:D
24375     { \c__tl_rescan_marker_tl \exp_not:N }
24376     #3 \scan_stop:
24377     \exp_after:wN \__tl_from_file_do:w
24378     \exp_after:wN \prg_do_nothing:
24379     \tex_input:D \l__tl_file_name_str \scan_stop:
24380     \exp_args:NNNo \group_end:
24381     #1 #2 \l__tl_internal_a_tl
24382   }
24383 }
24384 \exp_args:Nno \use:nn
24385 { \cs_new_protected:Npn \__tl_from_file_do:w #1 }
24386 { \c__tl_rescan_marker_tl }
24387 { \tl_set:No \l__tl_internal_a_tl {#1} }

```

(End definition for `\tl_set_from_file:Nnn` and others. These functions are documented on page 247.)

`\tl_set_from_file_x:Nnn` When reading a file and allowing expansion of the content, the set up only needs to prevent TeX complaining about the end of the file. That is done simply, with a group then used to trap the definition needed. Once the business is done using some scratch space, the tokens can be transferred to the real target.

```

\__tl_set_from_file_x:NNnn
24388 \cs_new_protected:Npn \tl_set_from_file_x:Nnn
24389 { \__tl_set_from_file_x:NNnn \tl_set:Nn }
24390 \cs_new_protected:Npn \tl_gset_from_file_x:Nnn
24391 { \__tl_set_from_file_x:NNnn \tl_gset:Nn }
24392 \cs_generate_variant:Nn \tl_set_from_file_x:Nnn { c }

```

```

24393 \cs_generate_variant:Nn \tl_gset_from_file_x:Nnn { c }
24394 \cs_new_protected:Npn \__tl_set_from_file_x:NNnn #1#2#3#4
24395 {
24396   \file_get_full_name:nN {#4} \l__tl_file_name_str
24397   \str_if_empty:NTF \l__tl_file_name_str
24398     { \__kernel_file_missing:n {#4} }
24399     {
24400       \group_begin:
24401       \etex_everyeof:D { \exp_not:N }
24402       #3 \scan_stop:
24403       \tl_set:Nx \l__tl_internal_a_tl
24404         { \tex_input:D \l__tl_file_name_str \c_space_token }
24405       \exp_args:NNNo \group_end:
24406       #1 #2 \l__tl_internal_a_tl
24407     }
24408 }

```

(End definition for `\tl_set_from_file_x:Nnn`, `\tl_gset_from_file_x:Nnn`, and `__tl_set_from_file_x:NNnn`. These functions are documented on page 248.)

`\l__tl_file_name_str`

```

24409 \str_new:N \l__tl_file_name_str

```

(End definition for `\l__tl_file_name_str`.)

43.13.1 Unicode case changing

The mechanisms needed for case changing are somewhat involved, particularly to allow for all of the special cases. These functions also require the appropriate data extracted from the Unicode documentation (either manually or automatically).

First, some code which “belongs” in `l3tokens` but has to come here.

```

24410 <@@=char>

```

`\char_lower_case:N` Expandable character generation is done using a two-part approach. First, see if the
`\char_upper_case:N` current character has a special mapping for the current transformation. If it does, insert
`\char_mixed_case:N` that. Otherwise, use the `TeX` data to look up the one-to-one mapping, and generate the
`\char_fold_case:N` appropriate character with the appropriate category code. Mixed case needs an extra step
`__char_change_case:nNN` as it may be special-cased or might be a special upper case outcome. The internal when
`__char_change_case:nN` using non-Unicode engines has to be set up to only do anything with ASCII characters.

```

24411 \cs_new:Npn \char_lower_case:N #1
24412 { \__char_change_case:nNN { lower } \char_value_lccode:n #1 }
24413 \cs_new:Npn \char_upper_case:N #1
24414 { \__char_change_case:nNN { upper } \char_value_uccode:n #1 }
24415 \cs_new:Npn \char_mixed_case:N #1
24416 {
24417   \tl_if_exist:cTF { c__char_mixed_case_ \token_to_str:N #1 _tl }
24418     { \tl_use:c { c__char_mixed_case_ \token_to_str:N #1 _tl } }
24419     { \char_upper_case:N #1 }
24420 }
24421 \cs_new:Npn \char_fold_case:N #1
24422 { \__char_change_case:nNN { fold } \char_value_lccode:n #1 }
24423 \cs_new:Npn \__char_change_case:nNN #1#2#3
24424 {
24425   \tl_if_exist:cTF { c__char_ #1 _case_ \token_to_str:N #3 _tl }

```

```

24426     { \tl_use:c { c__char_ #1 _case_ \token_to_str:N #3 _tl } }
24427     { \exp_args:Nf \__char_change_case:nN { #2 { ' #3 } } #3 }
24428   }
24429 \cs_new:Npn \__char_change_case:nN #1#2
24430 {
24431   \int_compare:nNnTF {#1} = 0
24432     {#2}
24433     { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
24434 }
24435 \bool_lazy_or:nnF { \sys_if_engine luatex_p: } { \sys_if_engine xetex_p: }
24436 {
24437   \cs_set_eq:NN \__char_change_case:nN \use_ii:nn
24438 }

```

(End definition for `\char_lower_case:N` and others. These functions are documented on page 250.)

`\char_codepoint_to_bytes:n`

```

\__char_codepoint_to_bytes_auxi:n
\__char_codepoint_to_bytes_auxii:Nnn
\__char_codepoint_to_bytes_auxiii:n

```

This code converts a codepoint into the correct UTF-8 representation. As there are a variable number of octets, the result starts with the numeral 1–4 to indicate the nature of the returned value. In terms of the algorithm itself, see <https://en.wikipedia.org/wiki/UTF-8> for the octet pattern.

```

24439 \cs_new:Npn \char_codepoint_to_bytes:n #1
24440 {
24441   \exp_args:Nf \__char_codepoint_to_bytes_auxi:n
24442     { \int_eval:n {#1} }
24443 }
24444 \cs_new:Npn \__char_codepoint_to_bytes_auxi:n #1
24445 {
24446   \if_int_compare:w #1 > "80 \exp_stop_f:
24447   \if_int_compare:w #1 < "800 \exp_stop_f:
24448     2
24449     \__char_codepoint_to_bytes_auxii:Nnn C {#1} { 64 }
24450     \__char_codepoint_to_bytes_auxiii:n {#1}
24451   \else:
24452     \if_int_compare:w #1 < "10000 \exp_stop_f:
24453       3
24454       \__char_codepoint_to_bytes_auxii:Nnn E {#1} { 64 * 64 }
24455       \__char_codepoint_to_bytes_auxiii:n
24456         { \int_div_truncate:nn {#1} { 64 } }
24457       \__char_codepoint_to_bytes_auxiii:n {#1}
24458     \else:
24459       4
24460       \__char_codepoint_to_bytes_auxii:Nnn F
24461         {#1} { 64 * 64 * 64 }
24462       \__char_codepoint_to_bytes_auxiii:n
24463         { \int_div_truncate:nn {#1} { 64 * 64 } }
24464       \__char_codepoint_to_bytes_auxiii:n
24465         { \int_div_truncate:nn {#1} { 64 } }
24466       \__char_codepoint_to_bytes_auxiii:n {#1}
24467     \fi:
24468   \fi:
24469   \else:
24470     1 {#1}
24471   \fi:
24472 }

```

```

24473 \cs_new:Npn \__char_codepoint_to_bytes_auxii:Nnn #1#2#3
24474 { { \int_eval:n { "#10 + \int_div_truncate:nn {#2} {#3} } } }
24475 \cs_new:Npn \__char_codepoint_to_bytes_auxiii:n #1
24476 { { \int_eval:n { \int_mod:nn {#1} { 64 } + 128 } } }

```

(End definition for `\char_codepoint_to_bytes:n` and others. This function is documented on page 250.)

```

24477 <@@=tl>

```

`\tl_if_head_eq_catcode:oNTF` Extra variants.

```

24478 \cs_generate_variant:Nn \tl_if_head_eq_catcode:nNTF { o }

```

(End definition for `\tl_if_head_eq_catcode:nNTF`. This function is documented on page 45.)

`\tl_lower_case:n` `\tl_upper_case:n` `\tl_mixed_case:n` The user level functions here are all wrappers around the internal functions for case changing.

```

24479 \cs_new:Npn \tl_lower_case:n { \__tl_change_case:nnn { lower } { } }
24480 \cs_new:Npn \tl_upper_case:n { \__tl_change_case:nnn { upper } { } }
24481 \cs_new:Npn \tl_mixed_case:n { \__tl_change_case:nnn { mixed } { } }
24482 \cs_new:Npn \tl_lower_case:nn { \__tl_change_case:nnn { lower } }
24483 \cs_new:Npn \tl_upper_case:nn { \__tl_change_case:nnn { upper } }
24484 \cs_new:Npn \tl_mixed_case:nn { \__tl_change_case:nnn { mixed } }

```

(End definition for `\tl_lower_case:n` and others. These functions are documented on page 244.)

The mechanism for the core conversion of case is based on the idea that we can use a loop to grab the entire token list plus a quark: the latter is used as an end marker and to avoid any brace stripping. Depending on the nature of the first item in the grabbed argument, it can either processed as a single token, treated as a group or treated as a space. These different cases all work by re-reading #1 in the appropriate way, hence the repetition of #1 `\q_recursion_stop`.

```

24485 \cs_new:Npn \__tl_change_case:nnn #1#2#3
24486 {
24487   \__kernel_exp_not:w \exp_after:wN
24488   {
24489     \exp:w
24490     \__tl_change_case_aux:nnn {#1} {#2} {#3}
24491   }
24492 }
24493 \cs_new:Npn \__tl_change_case_aux:nnn #1#2#3
24494 {
24495   \group_align_safe_begin:
24496   \__tl_change_case_loop:wnn
24497   #3 \q_recursion_tail \q_recursion_stop {#1} {#2}
24498   \__tl_change_case_result:n { }
24499 }
24500 \cs_new:Npn \__tl_change_case_loop:wnn #1 \q_recursion_stop
24501 {
24502   \tl_if_head_is_N_type:nTF {#1}
24503   { \__tl_change_case_N_type:Nwnn }
24504   {
24505     \tl_if_head_is_group:nTF {#1}
24506     { \__tl_change_case_group:nwnn }
24507     { \__tl_change_case_space:wnn }

```



```

24508     }
24509     #1 \q_recursion_stop
24510 }

```

Earlier versions of the code where only x-type expandable rather than f-type: this causes issues with nesting and so the slight performance hit is taken for a better outcome in usability terms. Setting up for f-type expandability has two requirements: a marker token after the main loop (see above) and a mechanism to “load” and finalise the result. That is handled in the code below, which includes the necessary material to end the \exp:w expansion.

```

24511 \cs_new:Npn \__tl_change_case_output:nwn #1#2 \__tl_change_case_result:n #3
24512 { #2 \__tl_change_case_result:n { #3 #1 } }
24513 \cs_generate_variant:Nn \__tl_change_case_output:nwn { V , o , v , f }
24514 \cs_new:Npn \__tl_change_case_end:wn #1 \__tl_change_case_result:n #2
24515 {
24516     \group_align_safe_end:
24517     \exp_end:
24518     #2
24519 }

```

Handling for the cases where the current argument is a brace group or a space is relatively easy. For the brace case, the routine works recursively, using the expandability of the mechanism to ensure that the result is finalised before storage. For the space case it is simply a question of removing the space in the input and storing it in the output. In both cases, and indeed for the N-type grabber, after removing the current item from the input __tl_change_case_loop:wnn is inserted in front of the remaining tokens.

```

24520 \cs_new:Npn \__tl_change_case_group:nwnn #1#2 \q_recursion_stop #3#4
24521 {
24522     \use:c { __tl_change_case_group_ #3 : nnnn } {#1} {#2} {#3} {#4}
24523 }
24524 \cs_new:Npn \__tl_change_case_group_lower:nnnn #1#2#3#4
24525 {
24526     \__tl_change_case_output:own
24527     {
24528         \exp_after:wN
24529         {
24530             \exp:w
24531             \__tl_change_case_aux:nnn {#3} {#4} {#1}
24532         }
24533     }
24534     \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
24535 }
24536 \cs_new_eq:NN \__tl_change_case_group_upper:nnnn
24537 \__tl_change_case_group_lower:nnnn

```

For the “mixed” case, a group is taken as forcing a switch to lower casing. That means we need a separate auxiliary. (Tracking whether we have found a first character inside a group and transferring the information out looks pretty horrible.)

```

24538 \cs_new:Npn \__tl_change_case_group_mixed:nnnn #1#2#3#4
24539 {
24540     \__tl_change_case_output:own
24541     {
24542         \exp_after:wN
24543         {

```

```

24544         \exp:w
24545         \__tl_change_case_aux:nnn {#3} {#4} {#1}
24546     }
24547 }
24548 \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#4}
24549 }
24550 \exp_last_unbraced:NNo \cs_new:Npn \__tl_change_case_space:wnn \c_space_tl
24551 {
24552     \__tl_change_case_output:nwn { ~ }
24553     \__tl_change_case_loop:wnn
24554 }

```

For N-type arguments there are several stages to the approach. First, a simply check for the end-of-input marker, which if found triggers the final clean up and output step. Assuming that is not the case, the first check is for math-mode escaping: this test can encompass control sequences or other N-type tokens so is handled up front.

```

24555 \cs_new:Npn \__tl_change_case_N_type:Nwnn #1#2 \q_recursion_stop
24556 {
24557     \quark_if_recursion_tail_stop_do:Nn #1
24558     { \__tl_change_case_end:wn }
24559     \exp_after:wN \__tl_change_case_N_type:NNNnnn
24560     \exp_after:wN #1 \l_tl_change_case_math_tl
24561     \q_recursion_tail ? \q_recursion_stop {#2}
24562 }

```

Looking for math mode escape first requires a loop over the possible token pairs to see if the current input (#1) matches an open-math case (#2). If it does then this test loop is ended and a new input-gathering one is begun. The latter simply transfers material from the input to the output without any expansion, testing each N-type token to see if it matches the close-math case required. If that is the situation then the “math loop” stops and resumes the main loop: as that might be either the standard case-changing one or the mixed-case alternative, it is not hard-coded into the math loop but is rather passed as argument #3 to `__tl_change_case_math:NNNnnn`. If no close-math token is found then the final clean-up is forced (*i.e.* there is no assumption of “well-behaved” input in terms of math mode).

```

24563 \cs_new:Npn \__tl_change_case_N_type:NNNnnn #1#2#3
24564 {
24565     \quark_if_recursion_tail_stop_do:Nn #2
24566     { \__tl_change_case_N_type:Nnnn #1 }
24567     \token_if_eq_meaning:NNTF #1 #2
24568     {
24569         \use_i_delimit_by_q_recursion_stop:nw
24570         {
24571             \__tl_change_case_math:NNNnnn
24572             #1 #3 \__tl_change_case_loop:wnn
24573         }
24574     }
24575     { \__tl_change_case_N_type:NNNnnn #1 }
24576 }
24577 \cs_new:Npn \__tl_change_case_math:NNNnnn #1#2#3#4
24578 {
24579     \__tl_change_case_output:nwn {#1}
24580     \__tl_change_case_math_loop:wNNnn #4 \q_recursion_stop #2 #3
24581 }

```

```

24582 \cs_new:Npn \__tl_change_case_math_loop:wNNnn #1 \q_recursion_stop
24583 {
24584   \tl_if_head_is_N_type:nTF {#1}
24585   { \__tl_change_case_math:NwNNnn }
24586   {
24587     \tl_if_head_is_group:nTF {#1}
24588     { \__tl_change_case_math_group:nwNNnn }
24589     { \__tl_change_case_math_space:wNNnn }
24590   }
24591   #1 \q_recursion_stop
24592 }
24593 \cs_new:Npn \__tl_change_case_math:NwNNnn #1#2 \q_recursion_stop #3#4
24594 {
24595   \token_if_eq_meaning:NNTF \q_recursion_tail #1
24596   { \__tl_change_case_end:wn }
24597   {
24598     \__tl_change_case_output:nwn {#1}
24599     \token_if_eq_meaning:NNTF #1 #3
24600     { #4 #2 \q_recursion_stop }
24601     { \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop #3#4 }
24602   }
24603 }
24604 \cs_new:Npn \__tl_change_case_math_group:nwNNnn #1#2 \q_recursion_stop
24605 {
24606   \__tl_change_case_output:nwn { {#1} }
24607   \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop
24608 }
24609 \exp_last_unbraced:NNo
24610 \cs_new:Npn \__tl_change_case_math_space:wNNnn \c_space_tl
24611 {
24612   \__tl_change_case_output:nwn { ~ }
24613   \__tl_change_case_math_loop:wNNnn
24614 }

```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: they cannot be used in the lookup table and also may require expansion. At this stage the loop code starting `__tl_change_case_loop:wnn` is inserted: all subsequent steps in the code which need a look-ahead are coded to rely on this and thus have w-type arguments if they may do a look-ahead.

```

24615 \cs_new:Npn \__tl_change_case_N_type:Nnnn #1#2#3#4
24616 {
24617   \token_if_cs:NNTF #1
24618   { \__tl_change_case_cs_letterlike:Nn #1 {#3} }
24619   { \use:c { \__tl_change_case_char_ #3 :Nnn } #1 {#3} {#4} }
24620   \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
24621 }

```

For character tokens there are some special cases to deal with then the majority of changes are covered by using the TeX data as a lookup along with expandable character generation. This avoids needing a very large number of macros or (as seen in earlier versions) a somewhat tricky split of the characters into various blocks. Notice that the special case code may do a look-ahead so requires a final w-type argument whereas the core lookup table does not and also guarantees an output so f-type expansion may be used to obtain the case-changed result.

```

24622 \cs_new:Npn \__tl_change_case_char_lower:Nnn #1#2#3
24623 {
24624   \cs_if_exist_use:cF { __tl_change_case_ #2 _ #3 :Nnw }
24625   { \use_ii:nn }
24626   #1
24627   {
24628     \use:c { __tl_change_case_ #2 _ sigma:Nnw } #1
24629     { \__tl_change_case_char:nN {#2} #1 }
24630   }
24631 }
24632 \cs_new_eq:NN \__tl_change_case_char_upper:Nnn
24633 \__tl_change_case_char_lower:Nnn

```

For mixed case, the code is somewhat different: there is a need to look up both mixed and upper case chars and we have to cover the situation where there is a character to skip over.

```

24634 \cs_new:Npn \__tl_change_case_char_mixed:Nnn #1#2#3
24635 {
24636   \__tl_change_case_mixed_switch:w
24637   \cs_if_exist_use:cF { __tl_change_case_mixed_ #3 :Nnw }
24638   {
24639     \cs_if_exist_use:cF { __tl_change_case_upper_ #3 :Nnw }
24640     { \use_ii:nn }
24641   }
24642   #1
24643   { \__tl_change_case_mixed_skip:N #1 }
24644 }

```

For Unicode engines we can handle all characters directly. However, for the 8-bit engines the aim is to deal with (a subset of) Unicode (UTF-8) input. They deal with that by making the upper half of the range active, so we look for that and if found work out how many UTF-8 octets there are to deal with. Those can then be grabbed to reconstruct the full Unicode character, which is then used in a lookup. (As will become obvious below, there is no intention here of covering all of Unicode.)

```

24645 \bool_lazy_or:nnTF
24646 { \sys_if_engine luatex_p: }
24647 { \sys_if_engine xetex_p: }
24648 {
24649   \cs_new:Npn \__tl_change_case_char:nN #1#2
24650   {
24651     \__tl_change_case_output:fwn
24652     { \use:c { char_ #1 _case:N } #2 }
24653   }
24654 }
24655 {
24656   \cs_new:Npn \__tl_change_case_char:nN #1#2
24657   {
24658     \int_compare:nNnTF { '#2 } > { "80 }
24659     {
24660       \int_compare:nNnTF { '#2 } < { "E0 }
24661       { \__tl_change_case_char_UTFviii:nNNN {#1} #2 }
24662       {
24663         \int_compare:nNnTF { '#2 } < { "F0 }
24664         { \__tl_change_case_char_UTFviii:nNNNN {#1} #2 }

```

```

24665         { \_tl_change_case_char_UTFviii:nNNNNN {#1} #2 }
24666     }
24667 }
24668 {
24669     \_tl_change_case_output:fwN
24670     { \use:c { char_ #1 _case:N } #2 }
24671 }
24672 }
24673 }

```

To allow for the special case of mixed case, we insert here a action-dependent auxiliary.

```

24674 \bool_lazy_or:nnF
24675 { \sys_if_engine_luatex_p: }
24676 { \sys_if_engine_xetex_p: }
24677 {
24678     \cs_new:Npn \_tl_change_case_char_UTFviii:nNNN #1#2#3#4
24679     { \_tl_change_case_char_UTFviii:nnN {#1} {#2#4} #3 }
24680     \cs_new:Npn \_tl_change_case_char_UTFviii:nNNNN #1#2#3#4#5
24681     { \_tl_change_case_char_UTFviii:nnN {#1} {#2#4#5} #3 }
24682     \cs_new:Npn \_tl_change_case_char_UTFviii:nNNNNN #1#2#3#4#5#6
24683     { \_tl_change_case_char_UTFviii:nnN {#1} {#2#4#5#6} #3 }
24684     \cs_new:Npn \_tl_change_case_char_UTFviii:nnN #1#2#3
24685     {
24686         \cs_if_exist:cTF { c__tl_ #1 _case_ \tl_to_str:n {#2} _tl }
24687         {
24688             \_tl_change_case_output:vwN
24689             { c__tl_ #1 _case_ \tl_to_str:n {#2} _tl }
24690         }
24691         { \_tl_change_case_output:nwn {#2} }
24692     } #3
24693 }
24694 }

```

Before dealing with general control sequences there are the special ones to deal with. Letter-like control sequences are a simple look-up, while for accents the loop is much as done elsewhere. Notice that we have a no-op test to make sure there is no unexpected expansion of letter-like input. The split into two parts here allows us to insert the “switch” code for mixed casing.

```

24695 \cs_new:Npn \_tl_change_case_cs_letterlike:Nn #1#2
24696 {
24697     \str_if_eq:nnTF {#2} { mixed }
24698     {
24699         \_tl_change_case_cs_letterlike:NnN #1 { upper }
24700         \_tl_change_case_mixed_switch:w
24701     }
24702     { \_tl_change_case_cs_letterlike:NnN #1 {#2} \prg_do_nothing: }
24703 }
24704 \cs_new:Npn \_tl_change_case_cs_letterlike:NnN #1#2#3
24705 {
24706     \cs_if_exist:cTF { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
24707     {
24708         \_tl_change_case_output:vwN
24709         { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
24710     } #3
24711 }

```

```

24712 {
24713   \cs_if_exist:cTF
24714   {
24715     c__tl_change_case_
24716     \str_if_eq:nnTF {#2} { lower } { upper } { lower }
24717     _ \token_to_str:N #1 _tl
24718   }
24719   {
24720     \__tl_change_case_output:nwn {#1}
24721     #3
24722   }
24723   {
24724     \exp_after:wN \__tl_change_case_cs_accents:NN
24725     \exp_after:wN #1 \l_tl_case_change_accents_tl
24726     \q_recursion_tail \q_recursion_stop
24727   }
24728 }
24729 }
24730 \cs_new:Npn \__tl_change_case_cs_accents:NN #1#2
24731 {
24732   \quark_if_recursion_tail_stop_do:Nn #2
24733   { \__tl_change_case_cs:N #1 }
24734   \str_if_eq:nnTF {#1} {#2}
24735   {
24736     \use_i_delimit_by_q_recursion_stop:nw
24737     { \__tl_change_case_output:nwn {#1} }
24738   }
24739   { \__tl_change_case_cs_accents:NN #1 }
24740 }

```

To deal with a control sequence there is first a need to test if it is on the list which indicate that case changing should be skipped. That's done using a loop as for the other special cases. If a hit is found then the argument is grabbed: that comes *after* the loop function which is therefore rearranged. In a L^AT_EX 2_ε context, `\protect` needs to be treated specially, to prevent expansion of the next token but output it without braces.

```

24741 \cs_new:Npn \__tl_change_case_cs:N #1
24742 {
24743   \*package
24744   \str_if_eq:nnTF {#1} { \protect } { \__tl_change_case_protect:wNN }
24745   \*package
24746   \exp_after:wN \__tl_change_case_cs:NN
24747   \exp_after:wN #1 \l_tl_case_change_exclude_tl
24748   \q_recursion_tail \q_recursion_stop
24749 }
24750 \cs_new:Npn \__tl_change_case_cs:NN #1#2
24751 {
24752   \quark_if_recursion_tail_stop_do:Nn #2
24753   {
24754     \__tl_change_case_cs_expand:Nnw #1
24755     { \__tl_change_case_output:nwn {#1} }
24756   }
24757   \str_if_eq:nnTF {#1} {#2}
24758   {
24759     \use_i_delimit_by_q_recursion_stop:nw

```

```

24760         { \_tl_change_case_cs:NNn #1 }
24761     }
24762     { \_tl_change_case_cs:NN #1 }
24763 }
24764 \cs_new:Npn \_tl_change_case_cs:NNn #1#2#3
24765 {
24766     \_tl_change_case_output:nwn { #1 {#3} }
24767     #2
24768 }
24769 (*package)
24770 \cs_new:Npn \_tl_change_case_protect:wNN #1 \q_recursion_stop #2 #3
24771 { \_tl_change_case_output:nwn { \protect #3 } #2 }
24772 (/package)

```

When a control sequence is not on the exclude list the other test if to see if it is expandable. Once again, if there is a hit then the loop function is grabbed as part of the clean-up and reinserted before the now expanded material. The test for expandability has to check for end-of-recursion as it is needed by the look-ahead code which might hit the end of the input. The test is done in two parts as `\bool_if:nTF` would choke if #1 was (!

```

24773 \cs_new:Npn \_tl_change_case_if_expandable:NTF #1
24774 {
24775     \token_if_expandable:NTF #1
24776     {
24777         \bool_lazy_any:nTF
24778         {
24779             { \token_if_eq_meaning_p:NN \q_recursion_tail #1 }
24780             { \token_if_protected_macro_p:N #1 }
24781             { \token_if_protected_long_macro_p:N #1 }
24782         }
24783         { \use_ii:nn }
24784         { \use_i:nn }
24785     }
24786     { \use_ii:nn }
24787 }
24788 \cs_new:Npn \_tl_change_case_cs_expand:Nnw #1#2
24789 {
24790     \_tl_change_case_if_expandable:NTF #1
24791     { \_tl_change_case_cs_expand:NN #1 }
24792     { #2 }
24793 }
24794 \cs_new:Npn \_tl_change_case_cs_expand:NN #1#2
24795 { \exp_after:wN #2 #1 }

```

For mixed case, there is an additional list of exceptions to deal with: once that is sorted, we can move on back to the main loop.

```

24796 \cs_new:Npn \_tl_change_case_mixed_skip:N #1
24797 {
24798     \exp_after:wN \_tl_change_case_mixed_skip:NN
24799     \exp_after:wN #1 \l_tl_mixed_case_ignore_tl
24800     \q_recursion_tail \q_recursion_stop
24801 }
24802 \cs_new:Npn \_tl_change_case_mixed_skip:NN #1#2
24803 {
24804     \quark_if_recursion_tail_stop_do:nn {#2}

```

```

24805     { \_tl_change_case_char:nN { mixed } #1 }
24806 \int_compare:nNnT { '#1 } = { '#2 }
24807     {
24808         \use_i_delimit_by_q_recursion_stop:nw
24809         {
24810             \_tl_change_case_output:nwn {#1}
24811             \_tl_change_case_mixed_skip_tidy:Nwn
24812         }
24813     }
24814     \_tl_change_case_mixed_skip:NN #1
24815 }
24816 \cs_new:Npn \_tl_change_case_mixed_skip_tidy:Nwn #1#2 \q_recursion_stop #3
24817 {
24818     \_tl_change_case_loop:wnn #2 \q_recursion_stop { mixed }
24819 }

```

Needed to switch from mixed to lower casing when we have found a first character in the former mode.

```

24820 \cs_new:Npn \_tl_change_case_mixed_switch:w
24821 #1 \_tl_change_case_loop:wnn #2 \q_recursion_stop #3
24822 {
24823     #1
24824     \_tl_change_case_loop:wnn #2 \q_recursion_stop { lower }
24825 }

```

(End definition for _tl_change_case:nnn and others.)

_tl_change_case_lower_sigma:Nnw If the current char is an upper case sigma, then a check is made on the next item in the input. If it is N-type and not a control sequence then there is a look-ahead phase.

```

\_tl_change_case_lower_sigma:w
\_tl_change_case_lower_sigma:Nw
\_tl_change_case_upper_sigma:Nnw
24826 \cs_new:Npn \_tl_change_case_lower_sigma:Nnw #1#2#3#4 \q_recursion_stop
24827 {
24828     \int_compare:nNnTF { '#1 } = { "03A3 }
24829     {
24830         \_tl_change_case_output:fwn
24831         { \_tl_change_case_lower_sigma:w #4 \q_recursion_stop }
24832     }
24833     {#2}
24834     #3 #4 \q_recursion_stop
24835 }
24836 \cs_new:Npn \_tl_change_case_lower_sigma:w #1 \q_recursion_stop
24837 {
24838     \tl_if_head_is_N_type:nTF {#1}
24839     { \_tl_change_case_lower_sigma:Nw #1 \q_recursion_stop }
24840     { \c_tl_final_sigma_tl }
24841 }
24842 \cs_new:Npn \_tl_change_case_lower_sigma:Nw #1#2 \q_recursion_stop
24843 {
24844     \_tl_change_case_if_expandable:Nw #1
24845     {
24846         \exp_after:wN \_tl_change_case_lower_sigma:w #1
24847         #2 \q_recursion_stop
24848     }
24849     {
24850         \token_if_letter:Nw #1
24851         { \c_tl_std_sigma_tl }

```



```

24852         { \c__tl_final_sigma_tl }
24853     }
24854 }

```

Simply skip to the final step for upper casing.

```

24855 \cs_new_eq:NN \__tl_change_case_upper_sigma:Nnw \use_ii:nn

```

(End definition for __tl_change_case_lower_sigma:Nnw and others.)

```

\__tl_change_case_lower_tr:Nnw
\__tl_change_case_lower_tr_auxi:Nw
\__tl_change_case_lower_tr_auxii:Nw
\__tl_change_case_upper_tr:Nnw
\__tl_change_case_lower_az:Nnw
\__tl_change_case_upper_az:Nnw

```

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

```

24856 \bool_lazy_or:nnTF
24857 { \sys_if_engine luatex_p: }
24858 { \sys_if_engine xetex_p: }
24859 {
24860     \cs_new:Npn \__tl_change_case_lower_tr:Nnw #1#2
24861     {
24862         \int_compare:nNnTF { '#1 } = { "0049 }
24863         { \__tl_change_case_lower_tr_auxi:Nw }
24864         {
24865             \int_compare:nNnTF { '#1 } = { "0130 }
24866             { \__tl_change_case_output:wnw { i } }
24867             { #2 }
24868         }
24869     }

```

After a dotless-I there may be a dot-above character. If there is then a dotted-i should be produced, otherwise output a dotless-i. When the combination is found both the dotless-I and the dot-above char have to be removed from the input, which is done by the \use_i:nn (it grabs __tl_change_case_loop:wn and the dot-above char and discards the latter).

```

24870 \cs_new:Npn \__tl_change_case_lower_tr_auxi:Nw #1#2 \q_recursion_stop
24871 {
24872     \tl_if_head_is_N_type:nTF {#2}
24873     { \__tl_change_case_lower_tr_auxii:Nw #2 \q_recursion_stop }
24874     { \__tl_change_case_output:Vwn \c__tl_dotless_i_tl }
24875     #1 #2 \q_recursion_stop
24876 }
24877 \cs_new:Npn \__tl_change_case_lower_tr_auxii:Nw #1#2 \q_recursion_stop
24878 {
24879     \__tl_change_case_if_expandable:NTF #1
24880     {
24881         \exp_after:wN \__tl_change_case_lower_tr_auxi:Nw #1
24882         #2 \q_recursion_stop
24883     }
24884     {
24885         \bool_lazy_or:nnTF
24886         { \token_if_cs_p:N #1 }
24887         { ! \int_compare_p:nNn { '#1 } = { "0307 } }
24888         { \__tl_change_case_output:Vwn \c__tl_dotless_i_tl }
24889         {
24890             \__tl_change_case_output:wnw { i }
24891             \use_i:nn
24892         }

```

```

24893     }
24894   }
24895 }

```

For 8-bit engines, dot-above is not available so there is a simple test for an upper-case I. Then we can look for the UTF-8 representation of an upper case dotted-I without the combining char. If it's not there, preserve the UTF-8 sequence as-is.

```

24896 {
24897   \cs_new:Npn \__tl_change_case_lower_tr:Nnw #1#2
24898   {
24899     \int_compare:nNnTF { '#1 } = { "0049 }
24900     { \__tl_change_case_output:Vwn \c__tl_dotless_i_tl }
24901     {
24902       \int_compare:nNnTF { '#1 } = { 196 }
24903       { \__tl_change_case_lower_tr_auxi:Nw #1 {#2} }
24904       {#2}
24905     }
24906   }
24907   \cs_new:Npn \__tl_change_case_lower_tr_auxi:Nw #1#2#3#4
24908   {
24909     \int_compare:nNnTF { '#4 } = { 176 }
24910     {
24911       \__tl_change_case_output:nwn { i }
24912       #3
24913     }
24914     {
24915       #2
24916       #3 #4
24917     }
24918   }
24919 }

```

Upper casing is easier: just one exception with no context.

```

24920 \cs_new:Npn \__tl_change_case_upper_tr:Nnw #1#2
24921 {
24922   \int_compare:nNnTF { '#1 } = { "0069 }
24923   { \__tl_change_case_output:Vwn \c__tl_dotted_I_tl }
24924   {#2}
24925 }

```

Straight copies.

```

24926 \cs_new_eq:NN \__tl_change_case_lower_az:Nnw \__tl_change_case_lower_tr:Nnw
24927 \cs_new_eq:NN \__tl_change_case_upper_az:Nnw \__tl_change_case_upper_tr:Nnw

```

(End definition for __tl_change_case_lower_tr:Nnw and others.)

```

\__tl_change_case_lower_lt:Nnw
\__tl_change_case_lower_lt:nNw
\__tl_change_case_lower_lt:nnw
\__tl_change_case_lower_lt:Nw
\__tl_change_case_lower_lt:NNw
\__tl_change_case_upper_lt:Nnw
\__tl_change_case_upper_lt:nnw
\__tl_change_case_upper_lt:Nw
\__tl_change_case_upper_lt:NNw

```

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. That means that there is some work to do when lower casing I and J. The first step is a simple match attempt: \c__tl_accents_lt_tl contains accented upper case letters which should gain a dot-above char in their lower case form. This is done using f-type expansion so only one pass is needed to find if it works or not. If there was no hit, the second stage is to check for I, J and I-ogonek, and if the current char is a match to look for a following accent.

```

24928 \cs_new:Npn \__tl_change_case_lower_lt:Nnw #1
24929 {

```

```

24930     \exp_args:Nf \_tl_change_case_lower_lt:nNnw
24931     { \str_case:nVF #1 \c__tl_accents_lt_tl \exp_stop_f: }
24932     #1
24933   }
24934 \cs_new:Npn \_tl_change_case_lower_lt:nNnw #1#2
24935 {
24936   \tl_if_blank:nTF {#1}
24937   {
24938     \exp_args:Nf \_tl_change_case_lower_lt:nnw
24939     {
24940       \int_case:nnF {'#2}
24941       {
24942         { "0049 } i
24943         { "004A } j
24944         { "012E } \c__tl_i_ogonek_tl
24945       }
24946       \exp_stop_f:
24947     }
24948   }
24949   {
24950     \_tl_change_case_output:wnw {#1}
24951     \use_none:n
24952   }
24953 }
24954 \cs_new:Npn \_tl_change_case_lower_lt:nnw #1#2
24955 {
24956   \tl_if_blank:nTF {#1}
24957   {#2}
24958   {
24959     \_tl_change_case_output:wnw {#1}
24960     \_tl_change_case_lower_lt:Nw
24961   }
24962 }

```

Grab the next char and see if it is one of the accents used in Lithuanian: if it is, add the dot-above char into the output.

```

24963 \cs_new:Npn \_tl_change_case_lower_lt:Nw #1#2 \q_recursion_stop
24964 {
24965   \tl_if_head_is_N_type:nT {#2}
24966   { \_tl_change_case_lower_lt:NNw }
24967   #1 #2 \q_recursion_stop
24968 }
24969 \cs_new:Npn \_tl_change_case_lower_lt:NNw #1#2#3 \q_recursion_stop
24970 {
24971   \_tl_change_case_if_expandable:NTF #2
24972   {
24973     \exp_after:wN \_tl_change_case_lower_lt:Nw \exp_after:wN #1 #2
24974     #3 \q_recursion_stop
24975   }
24976   {
24977     \bool_lazy_and:nnT
24978     { ! \token_if_cs_p:N #2 }
24979     {
24980       \bool_lazy_any_p:n

```

```

24981         {
24982             { \int_compare_p:nNn { '#2 } = { "0300 } }
24983             { \int_compare_p:nNn { '#2 } = { "0301 } }
24984             { \int_compare_p:nNn { '#2 } = { "0303 } }
24985         }
24986     }
24987     { \__tl_change_case_output:Vwn \c__tl_dot_above_tl }
24988     #1 #2#3 \q_recursion_stop
24989 }
24990 }

```

For upper casing, the test required is for a dot-above char after an I, J or I-ogonek. First a test for the appropriate letter, and if found a look-ahead and potentially one token dropped.

```

24991 \cs_new:Npn \__tl_change_case_upper_lt:Nnw #1
24992 {
24993     \exp_args:Nf \__tl_change_case_upper_lt:nnw
24994     {
24995         \int_case:nnF { '#1 }
24996         {
24997             { "0069 } I
24998             { "006A } J
24999             { "012F } \c__tl_I_ogonek_tl
25000         }
25001         \exp_stop_f:
25002     }
25003 }
25004 \cs_new:Npn \__tl_change_case_upper_lt:nnw #1#2
25005 {
25006     \tl_if_blank:nTF {#1}
25007     {#2}
25008     {
25009         \__tl_change_case_output:nnw {#1}
25010         \__tl_change_case_upper_lt:Nw
25011     }
25012 }
25013 \cs_new:Npn \__tl_change_case_upper_lt:Nw #1#2 \q_recursion_stop
25014 {
25015     \tl_if_head_is_N_type:nT {#2}
25016     { \__tl_change_case_upper_lt:NNw }
25017     #1 #2 \q_recursion_stop
25018 }
25019 \cs_new:Npn \__tl_change_case_upper_lt:NNw #1#2#3 \q_recursion_stop
25020 {
25021     \__tl_change_case_if_expandable:NTF #2
25022     {
25023         \exp_after:wN \__tl_change_case_upper_lt:Nw \exp_after:wN #1 #2
25024         #3 \q_recursion_stop
25025     }
25026     {
25027         \bool_lazy_and:nnTF
25028         { ! \token_if_cs_p:N #2 }
25029         { \int_compare_p:nNn { '#2 } = { "0307 } }
25030         { #1 }

```

```

25031         { #1 #2 }
25032         #3 \q_recursion_stop
25033     }
25034 }

```

(End definition for _tl_change_case_lower_lt:Nnw and others.)

_tl_change_case_upper_de-alt:Nnw A simple alternative version for German.

```

25035 \cs_new:cpn { \_tl_change_case_upper_de-alt:Nnw } #1#2
25036 {
25037     \int_compare:nNnTF { '#1 } = { 223 }
25038     { \_tl_change_case_output:Vwn \c__tl_upper_Eszett_tl }
25039     {#2}
25040 }

```

(End definition for _tl_change_case_upper_de-alt:Nnw.)

\c__tl_std_sigma_tl The above needs various special token lists containing pre-formed characters. This set
\c__tl_final_sigma_tl are only available in Unicode engines, with no-op definitions for 8-bit use.
\c__tl_accents_lt_tl
\c__tl_dot_above_tl
\c__tl_upper_Eszett_tl

```

25041 \bool_lazy_or:nnTF
25042 { \sys_if_engine luatex_p: }
25043 { \sys_if_engine xetex_p: }
25044 {
25045     \group_begin:
25046     \cs_set:Npn \_tl_tmp:n #1
25047     { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
25048     \tl_const:Nx \c__tl_std_sigma_tl { \_tl_tmp:n { "03C3 } }
25049     \tl_const:Nx \c__tl_final_sigma_tl { \_tl_tmp:n { "03C2 } }
25050     \tl_const:Nx \c__tl_accents_lt_tl
25051     {
25052         \_tl_tmp:n { "00CC }
25053         {
25054             \_tl_tmp:n { "0069 }
25055             \_tl_tmp:n { "0307 }
25056             \_tl_tmp:n { "0300 }
25057         }
25058         \_tl_tmp:n { "00CD }
25059         {
25060             \_tl_tmp:n { "0069 }
25061             \_tl_tmp:n { "0307 }
25062             \_tl_tmp:n { "0301 }
25063         }
25064         \_tl_tmp:n { "0128 }
25065         {
25066             \_tl_tmp:n { "0069 }
25067             \_tl_tmp:n { "0307 }
25068             \_tl_tmp:n { "0303 }
25069         }
25070     }
25071     \tl_const:Nx \c__tl_dot_above_tl { \_tl_tmp:n { "0307 } }
25072     \tl_const:Nx \c__tl_upper_Eszett_tl { \_tl_tmp:n { "1E9E } }
25073     \group_end:
25074 }
25075 {

```

```

25076 \tl_const:Nn \c__tl_std_sigma_tl { }
25077 \tl_const:Nn \c__tl_final_sigma_tl { }
25078 \tl_const:Nn \c__tl_accents_lt_tl { }
25079 \tl_const:Nn \c__tl_dot_above_tl { }
25080 \tl_const:Nn \c__tl_upper_Eszett_tl { }
25081 }

```

(End definition for \c__tl_std_sigma_tl and others.)

\c__tl_dotless_i_tl For cases where there is an 8-bit option in the T1 font set up, a variant is provided in both cases.

```

\c__tl_dotted_I_tl
\c__tl_i_ogonek_tl
\c__tl_I_ogonek_tl
25082 \group_begin:
25083 \bool_lazy_or:nnTF
25084 { \sys_if_engine_luatex_p: }
25085 { \sys_if_engine_xetex_p: }
25086 {
25087 \cs_set_protected:Npn \__tl_tmp:w #1#2
25088 {
25089 \tl_const:Nx #1
25090 { \char_generate:nn {"#2} { \char_value_catcode:n {"#2} } }
25091 }
25092 }
25093 {
25094 \cs_set_protected:Npn \__tl_tmp:w #1#2
25095 {
25096 \group_begin:
25097 \cs_set_protected:Npn \__tl_tmp:w ##1##2##3
25098 {
25099 \tl_const:Nx #1
25100 {
25101 \exp_after:wN \exp_after:wN \exp_after:wN
25102 \exp_not:N \char_generate:nn {##2} { 13 }
25103 \exp_after:wN \exp_after:wN \exp_after:wN
25104 \exp_not:N \char_generate:nn {##3} { 13 }
25105 }
25106 }
25107 \tl_set:Nx \l__tl_internal_a_tl
25108 { \char_codepoint_to_bytes:n {"#2} }
25109 \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
25110 \group_end:
25111 }
25112 }
25113 \__tl_tmp:w \c__tl_dotless_i_tl { 0131 }
25114 \__tl_tmp:w \c__tl_dotted_I_tl { 0130 }
25115 \__tl_tmp:w \c__tl_i_ogonek_tl { 012F }
25116 \__tl_tmp:w \c__tl_I_ogonek_tl { 012E }
25117 \group_end:

```

(End definition for \c__tl_dotless_i_tl and others.)

For 8-bit engines we now need to define the case-change data for the multi-octet mappings. These need a list of what code points are doable in T1 so the list is hard coded (there's no saving in loading the mappings dynamically). All of the straight-forward ones have two octets, so that is taken as read.

```

25118 \group_begin:

```

```

25119 \bool_lazy_or:nnT
25120 { \sys_if_engine_pdftex_p: }
25121 { \sys_if_engine_uptex_p: }
25122 {
25123   \cs_set_protected:Npn \__tl_loop:nn #1#2
25124   {
25125     \quark_if_recursion_tail_stop:n {#1}
25126     \tl_set:Nx \l__tl_internal_a_tl
25127     {
25128       \char_codepoint_to_bytes:n {"#1}
25129       \char_codepoint_to_bytes:n {"#2}
25130     }
25131     \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
25132     \__tl_loop:nn
25133   }
25134   \cs_set_protected:Npn \__tl_tmp:w #1#2#3#4#5#6
25135   {
25136     \tl_const:cx
25137     {
25138       c__tl_lower_case_
25139       \char_generate:nn {#2} { 12 }
25140       \char_generate:nn {#3} { 12 }
25141       _tl
25142     }
25143     {
25144       \exp_after:wN \exp_after:wN \exp_after:wN
25145       \exp_not:N \char_generate:nn {#5} { 13 }
25146       \exp_after:wN \exp_after:wN \exp_after:wN
25147       \exp_not:N \char_generate:nn {#6} { 13 }
25148     }
25149     \tl_const:cx
25150     {
25151       c__tl_upper_case_
25152       \char_generate:nn {#5} { 12 }
25153       \char_generate:nn {#6} { 12 }
25154       _tl
25155     }
25156     {
25157       \exp_after:wN \exp_after:wN \exp_after:wN
25158       \exp_not:N \char_generate:nn {#2} { 13 }
25159       \exp_after:wN \exp_after:wN \exp_after:wN
25160       \exp_not:N \char_generate:nn {#3} { 13 }
25161     }
25162   }
25163   \__tl_loop:nn
25164   { 00C0 } { 00E0 }
25165   { 00C2 } { 00E2 }
25166   { 00C3 } { 00E3 }
25167   { 00C4 } { 00E4 }
25168   { 00C5 } { 00E5 }
25169   { 00C6 } { 00E6 }
25170   { 00C7 } { 00E7 }
25171   { 00C8 } { 00E8 }
25172   { 00C9 } { 00E9 }

```

25173	{ 00CA }	{ 00EA }
25174	{ 00CB }	{ 00EB }
25175	{ 00CC }	{ 00EC }
25176	{ 00CD }	{ 00ED }
25177	{ 00CE }	{ 00EE }
25178	{ 00CF }	{ 00EF }
25179	{ 00D0 }	{ 00F0 }
25180	{ 00D1 }	{ 00F1 }
25181	{ 00D2 }	{ 00F2 }
25182	{ 00D3 }	{ 00F3 }
25183	{ 00D4 }	{ 00F4 }
25184	{ 00D5 }	{ 00F5 }
25185	{ 00D6 }	{ 00F6 }
25186	{ 00D8 }	{ 00F8 }
25187	{ 00D9 }	{ 00F9 }
25188	{ 00DA }	{ 00FA }
25189	{ 00DB }	{ 00FB }
25190	{ 00DC }	{ 00FC }
25191	{ 00DD }	{ 00FD }
25192	{ 00DE }	{ 00FE }
25193	{ 0100 }	{ 0101 }
25194	{ 0102 }	{ 0103 }
25195	{ 0104 }	{ 0105 }
25196	{ 0106 }	{ 0107 }
25197	{ 0108 }	{ 0109 }
25198	{ 010A }	{ 010B }
25199	{ 010C }	{ 010D }
25200	{ 010E }	{ 010F }
25201	{ 0110 }	{ 0111 }
25202	{ 0112 }	{ 0113 }
25203	{ 0114 }	{ 0115 }
25204	{ 0116 }	{ 0117 }
25205	{ 0118 }	{ 0119 }
25206	{ 011A }	{ 011B }
25207	{ 011C }	{ 011D }
25208	{ 011E }	{ 011F }
25209	{ 0120 }	{ 0121 }
25210	{ 0122 }	{ 0123 }
25211	{ 0124 }	{ 0125 }
25212	{ 0128 }	{ 0129 }
25213	{ 012A }	{ 012B }
25214	{ 012C }	{ 012D }
25215	{ 012E }	{ 012F }
25216	{ 0132 }	{ 0133 }
25217	{ 0134 }	{ 0135 }
25218	{ 0136 }	{ 0137 }
25219	{ 0139 }	{ 013A }
25220	{ 013B }	{ 013C }
25221	{ 013E }	{ 013F }
25222	{ 0141 }	{ 0142 }
25223	{ 0143 }	{ 0144 }
25224	{ 0145 }	{ 0146 }
25225	{ 0147 }	{ 0148 }
25226	{ 014A }	{ 014B }


```

25227 { 014C } { 014D }
25228 { 014E } { 014F }
25229 { 0150 } { 0151 }
25230 { 0152 } { 0153 }
25231 { 0154 } { 0155 }
25232 { 0156 } { 0157 }
25233 { 0158 } { 0159 }
25234 { 015A } { 015B }
25235 { 015C } { 015D }
25236 { 015E } { 015F }
25237 { 0160 } { 0161 }
25238 { 0162 } { 0163 }
25239 { 0164 } { 0165 }
25240 { 0168 } { 0169 }
25241 { 016A } { 016B }
25242 { 016C } { 016D }
25243 { 016E } { 016F }
25244 { 0170 } { 0171 }
25245 { 0172 } { 0173 }
25246 { 0174 } { 0175 }
25247 { 0176 } { 0177 }
25248 { 0178 } { 00FF }
25249 { 0179 } { 017A }
25250 { 017B } { 017C }
25251 { 017D } { 017E }
25252 { 01CD } { 01CE }
25253 { 01CF } { 01D0 }
25254 { 01D1 } { 01D2 }
25255 { 01D3 } { 01D4 }
25256 { 01E2 } { 01E3 }
25257 { 01E6 } { 01E7 }
25258 { 01E8 } { 01E9 }
25259 { 01EA } { 01EB }
25260 { 01F4 } { 01F5 }
25261 { 0218 } { 0219 }
25262 { 021A } { 021B }
25263 \q_recursion_tail ?
25264 \q_recursion_stop
25265 \cs_set_protected:Npn \__tl_tmp:w #1#2#3
25266 {
25267   \group_begin:
25268   \cs_set_protected:Npn \__tl_tmp:w ##1##2##3
25269   {
25270     \tl_const:cx
25271     {
25272       c__tl_ #3 _case_
25273       \char_generate:nn {##2} { 12 }
25274       \char_generate:nn {##3} { 12 }
25275       _tl
25276     }
25277     {#2}
25278   }
25279   \tl_set:Nx \l__tl_internal_a_tl
25280   { \char_codepoint_to_bytes:n { "#1 } }

```

```

25281         \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
25282     \group_end:
25283     }
25284     \__tl_tmp:w { 00DF } { SS } { upper }
25285     \__tl_tmp:w { 00DF } { Ss } { mixed }
25286     \__tl_tmp:w { 0131 } { I } { upper }
25287 }
25288 \group_end:

```

The (fixed) look-up mappings for letter-like control sequences.

```

25289 \group_begin:
25290 \cs_set_protected:Npn \__tl_change_case_setup:NN #1#2
25291 {
25292     \quark_if_recursion_tail_stop:N #1
25293     \tl_const:cn { c__tl_change_case_lower_ \token_to_str:N #1 _tl }
25294     { #2 }
25295     \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N #2 _tl }
25296     { #1 }
25297     \__tl_change_case_setup:NN
25298 }
25299 \__tl_change_case_setup:NN
25300 \AA \aa
25301 \AE \ae
25302 \DH \dh
25303 \DJ \dj
25304 \IJ \ij
25305 \L \l
25306 \NG \ng
25307 \O \o
25308 \OE \oe
25309 \SS \ss
25310 \TH \th
25311 \q_recursion_tail ?
25312 \q_recursion_stop
25313 \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \i _tl } { I }
25314 \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \j _tl } { J }
25315 \group_end:

```

\l_tl_case_change_accents_tl A list of accents to leave alone.

```

25316 \tl_new:N \l_tl_case_change_accents_tl
25317 \tl_set:Nn \l_tl_case_change_accents_tl
25318 { \" \' \. \^ \' \~ \c \H \k \r \t \u \v }

```

(End definition for \l_tl_case_change_accents_tl. This variable is documented on page 246.)

__tl_change_case_mixed_n1:Nnw For Dutch, there is a single look-ahead test for ij when title casing. If the appropriate letters are found, produce IJ and gobble the j/J.

```

\__tl_change_case_mixed_n1:Nw
\__tl_change_case_mixed_n1:Nnw
25319 \cs_new:Npn \__tl_change_case_mixed_n1:Nnw #1
25320 {
25321     \bool_lazy_or:nnTF
25322     { \int_compare_p:nNn { '#1 } = { 'i } }
25323     { \int_compare_p:nNn { '#1 } = { 'I } }
25324     {
25325         \__tl_change_case_output:nwn { I }

```

```

25326     \_tl_change_case_mixed_n1:Nw
25327   }
25328 }
25329 \cs_new:Npn \_tl_change_case_mixed_n1:Nw #1#2 \q_recursion_stop
25330 {
25331   \tl_if_head_is_N_type:nT {#2}
25332   { \_tl_change_case_mixed_n1:NNw }
25333   #1 #2 \q_recursion_stop
25334 }
25335 \cs_new:Npn \_tl_change_case_mixed_n1:NNw #1#2#3 \q_recursion_stop
25336 {
25337   \_tl_change_case_if_expandable:NTF #2
25338   {
25339     \exp_after:wN \_tl_change_case_mixed_n1:Nw \exp_after:wN #1 #2
25340     #3 \q_recursion_stop
25341   }
25342   {
25343     \bool_lazy_and:nnTF
25344     { ! ( \token_if_cs_p:N #2 ) }
25345     {
25346       \bool_lazy_or_p:nn
25347       { \int_compare_p:nNn { '#2 } = { 'j } }
25348       { \int_compare_p:nNn { '#2 } = { 'J } }
25349     }
25350     {
25351       \_tl_change_case_output:nwn { J }
25352       #1
25353     }
25354     { #1 #2 }
25355     #3 \q_recursion_stop
25356   }
25357 }

```

(End definition for `_tl_change_case_mixed_n1:Nnw`, `_tl_change_case_mixed_n1:Nw`, and `_tl_change_case_mixed_n1:NNw`.)

`\l_tl_case_change_math_tl` The list of token pairs which are treated as math mode and so not case changed.

```

25358 \tl_new:N \l_tl_case_change_math_tl
25359 \*package
25360 \tl_set:Nn \l_tl_case_change_math_tl
25361 { $ $ \ ( \ ) }
25362 \*package

```

(End definition for `\l_tl_case_change_math_tl`. This variable is documented on page 245.)

`\l_tl_case_change_exclude_tl` The list of commands for which an argument is not case changed.

```

25363 \tl_new:N \l_tl_case_change_exclude_tl
25364 \*package
25365 \tl_set:Nn \l_tl_case_change_exclude_tl
25366 { \cite \ensuremath \label \ref }
25367 \*package

```

(End definition for `\l_tl_case_change_exclude_tl`. This variable is documented on page 246.)

`\l_tl_mixed_case_ignore_tl` Characters to skip over when finding the first letter in a word to be mixed cased.

```

25368 \tl_new:N \l_tl_mixed_case_ignore_tl
25369 \tl_set:Nx \l_tl_mixed_case_ignore_tl
25370 {
25371   ( % )
25372   [ % ]
25373   \cs_to_str:N \{ % \}
25374   '
25375   -
25376 }

```

(End definition for `\l_tl_mixed_case_ignore_tl`. This variable is documented on page 247.)

43.13.2 Building a token list

Between `\tl_build_begin:N <tl var>` and `\tl_build_end:N <tl var>`, the `<tl var>` has the structure

```

\exp_end: ... \exp_end: \__tl_build_last:NNn <assignment> <next tl>
{<left>} <right>

```

where `<right>` is not braced. The “data” it represents is `<left>` followed by the “data” of `<next tl>` followed by `<right>`. The `<next tl>` is a token list variable whose name is that of `<tl var>` followed by `'`. There are between 0 and 4 `\exp_end:` to keep track of when `<left>` and `<right>` should be put into the `<next tl>`. The `<assignment>` is `\cs_set_nopar:Npx` if the variable is local, and `\cs_gset_nopar:Npx` if it is global.

```

\tl_build_begin:N
\tl_build_gbegin:N
\__tl_build_begin:NN
\__tl_build_begin:NNN

```

First construct the `<next tl>`: using a prime here conflicts with the usual `expl3` convention but we need a name that can be derived from `#1` without any external data such as a counter. Empty that `<next tl>` and setup the structure. The local and global versions only differ by a single function `\cs_(g)set_nopar:Npx` used for all assignments: this is important because only that function is stored in the `<tl var>` and `<next tl>` for subsequent assignments. In principle `__tl_build_begin:NNN` could use `\tl_(g)clear_new:N` to empty `#1` and make sure it is defined, but logging the definition does not seem useful so we just do `#3 #1 {}` to clear it locally or globally as appropriate.

```

25377 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
25378 \cs_new_protected:Npn \tl_build_begin:N #1
25379 { \__tl_build_begin:NN \cs_set_nopar:Npx #1 }
25380 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
25381 \cs_new_protected:Npn \tl_build_gbegin:N #1
25382 { \__tl_build_begin:NN \cs_gset_nopar:Npx #1 }
25383 \cs_new_protected:Npn \__tl_build_begin:NN #1#2
25384 { \exp_args:Nc \__tl_build_begin:NNN { \cs_to_str:N #2 ' } #2 #1 }
25385 \cs_new_protected:Npn \__tl_build_begin:NNN #1#2#3
25386 {
25387   #3 #1 { }
25388   #3 #2
25389   {
25390     \exp_not:n { \exp_end: \exp_end: \exp_end: \exp_end: }
25391     \exp_not:n { \__tl_build_last:NNn #3 #1 { } }
25392   }
25393 }

```

(End definition for `\tl_build_begin:N` and others. These functions are documented on page 249.)

`\tl_build_clear:N` The `begin` and `gbegin` functions already clear enough to make the token list variable effectively empty. Eventually the `begin` and `gbegin` functions should check that `#1` is empty or undefined, while the `clear` and `gclear` functions ought to empty `#1`, `#1''` and so on, similar to `\tl_build_end:N`. This only affects memory usage.

```
25394 \cs_new_eq:NN \tl_build_clear:N \tl_build_begin:N
25395 \cs_new_eq:NN \tl_build_gclear:N \tl_build_gbegin:N
```

(End definition for `\tl_build_clear:N` and `\tl_build_gclear:N`. These functions are documented on page 249.)

`\tl_build_put_right:Nn` Similar to `\tl_put_right:Nn`, but apply `\exp:w` to `#1`. Most of the time this just removes one `\exp_end:`. When there are none left, `__tl_build_last:NNn` is expanded instead.

`\tl_build_put_right:Nx` It resets the definition of the `\tl var` by ending the `\exp_not:n` and the definition early.

`\tl_build_gput_right:Nn` Then it makes sure the `\next tl` (its argument `#1`) is set-up and starts a new definition.

`\tl_build_gput_right:Nx` Then `__tl_build_put:nn` and `__tl_build_put:nw` place the `\left` part of the original `\tl var` as appropriate for the definition of the `\next tl` (the `\right` part is left in the right place without ever becoming a macro argument). We use `\exp_after:wN` rather than some `\exp_args:No` to avoid reading arguments that are likely very long token lists. We use `\cs_(g)set_nopar:Npx` rather than `\tl_(g)set:Nx` partly for the same reason and partly because the assignments are interrupted by brace tricks, which implies that the assignment does not simply set the token list to an x-expansion of the second argument.

```
25396 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
25397 \cs_new_protected:Npn \tl_build_put_right:Nn #1#2
25398 {
25399   \cs_set_nopar:Npx #1
25400   { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
25401 }
25402 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
25403 \cs_new_protected:Npn \tl_build_put_right:Nx #1#2
25404 {
25405   \cs_set_nopar:Npx #1
25406   { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
25407 }
25408 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
25409 \cs_new_protected:Npn \tl_build_gput_right:Nn #1#2
25410 {
25411   \cs_gset_nopar:Npx #1
25412   { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
25413 }
25414 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
25415 \cs_new_protected:Npn \tl_build_gput_right:Nx #1#2
25416 {
25417   \cs_gset_nopar:Npx #1
25418   { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
25419 }
25420 \cs_new_protected:Npn \__tl_build_last:NNn #1#2
25421 {
25422   \if_false: { { \fi:
25423     \exp_end: \exp_end: \exp_end: \exp_end:
25424     \__tl_build_last:NNn #1 #2 { }
25425   }
25426 }
25427 \if_meaning:w \c_empty_tl #2
```

```

25428     \__tl_build_begin:NN #1 #2
25429     \fi:
25430     #1 #2
25431     {
25432         \exp_after:wN \exp_not:n \exp_after:wN
25433         {
25434             \exp:w \if_false: } } \fi:
25435             \exp_after:wN \__tl_build_put:nn \exp_after:wN {#2}
25436         }
25437     \cs_new_protected:Npn \__tl_build_put:nn #1#2 { \__tl_build_put:nw {#2} #1 }
25438     \cs_new_protected:Npn \__tl_build_put:nw #1#2 \__tl_build_last:NNn #3#4#5
25439     { #2 \__tl_build_last:NNn #3 #4 { #1 #5 } }

```

(End definition for `\tl_build_put_right:Nn` and others. These functions are documented on page 250.)

`\tl_build_put_left:Nn` See `\tl_build_put_right:Nn` for all the machinery. We could easily provide `\tl_build_put_left_right:NNn`, by just add the *<right>* material after the *<left>* in the x-expanding assignment.

```

\__tl_build_put_left:Nn
\__tl_build_put_left:Nx
\tl_build_gput_left:Nn
\tl_build_gput_left:Nx
\__tl_build_put_left:NNn
25440 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
25441 \cs_new_protected:Npn \tl_build_put_left:Nn #1
25442 { \__tl_build_put_left:NNn \cs_set_nopar:Npx #1 }
25443 \cs_generate_variant:Nn \tl_build_put_left:Nn { Nx }
25444 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
25445 \cs_new_protected:Npn \tl_build_gput_left:Nn #1
25446 { \__tl_build_put_left:NNn \cs_gset_nopar:Npx #1 }
25447 \cs_generate_variant:Nn \tl_build_gput_left:Nn { Nx }
25448 \cs_new_protected:Npn \__tl_build_put_left:NNn #1#2#3
25449 {
25450     #1 #2
25451     {
25452         \exp_after:wN \exp_not:n \exp_after:wN
25453         {
25454             \exp:w \exp_after:wN \__tl_build_put:nn
25455             \exp_after:wN {#2} {#3}
25456         }
25457     }
25458 }

```

(End definition for `\tl_build_put_left:Nn`, `\tl_build_gput_left:Nn`, and `__tl_build_put_left:NNn`. These functions are documented on page 250.)

`\tl_build_get:NN` The idea is to expand the *<tl var>* then the *<next tl>* and so on, all within an x-expanding assignment, and wrap as appropriate in `\exp_not:n`. The various *<left>* parts are left in the assignment as we go, which enables us to expand the *<next tl>* at the right place. The various *<right>* parts are eventually picked up in one last `\exp_not:n`, with a brace trick to wrap all the *<right>* parts together.

```

25459 \cs_new_protected:Npn \tl_build_get:NN
25460 { \__tl_build_get:NNN \tl_set:Nx }
25461 \cs_new_protected:Npn \__tl_build_get:NNN #1#2#3
25462 { #1 #3 { \if_false: { \fi: \exp_after:wN \__tl_build_get:w #2 } } }
25463 \cs_new:Npn \__tl_build_get:w #1 \__tl_build_last:NNn #2#3#4
25464 {
25465     \exp_not:n {#4}
25466     \if_meaning:w \c_empty_tl #3

```

```

25467     \exp_after:wN \__tl_build_get_end:w
25468     \fi:
25469     \exp_after:wN \__tl_build_get:w #3
25470   }
25471 \cs_new:Npn \__tl_build_get_end:w #1#2#3
25472 { \exp_after:wN \exp_not:n \exp_after:wN { \if_false: } \fi: }

```

(End definition for `\tl_build_get:NN` and others. This function is documented on page 250.)

`\tl_build_end:N` Get the data then clear the *(next tl)* recursively until finding an empty one. It is perhaps wasteful to repeatedly use `\cs_to_sr:N`. The local/global scope is checked by `\tl_set:Nx` or `\tl_gset:Nx`.

```

25473 \cs_new_protected:Npn \tl_build_end:N #1
25474 {
25475   \__tl_build_get:NNN \tl_set:Nx #1 #1
25476   \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_clear:N
25477 }
25478 \cs_new_protected:Npn \tl_build_gend:N #1
25479 {
25480   \__tl_build_get:NNN \tl_gset:Nx #1 #1
25481   \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_gclear:N
25482 }
25483 \cs_new_protected:Npn \__tl_build_end_loop:NN #1#2
25484 {
25485   \if_meaning:w \c_empty_tl #1
25486     \exp_after:wN \use_none:nnnnnn
25487   \fi:
25488   #2 #1
25489   \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } #2
25490 }

```

(End definition for `\tl_build_end:N`, `\tl_build_gend:N`, and `__tl_build_end_loop:NN`. These functions are documented on page 250.)

43.13.3 Other additions to `\l3tl`

`\tl_rand_item:n` Importantly `\tl_item:nn` only evaluates its argument once.

```

\tl_rand_item:N 25491 \cs_new:Npn \tl_rand_item:n #1
\tl_rand_item:c 25492 {
25493   \tl_if_blank:nF {#1}
25494     { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
25495   }
25496 \cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
25497 \cs_generate_variant:Nn \tl_rand_item:N { c }

```

(End definition for `\tl_rand_item:n` and `\tl_rand_item:N`. These functions are documented on page 248.)

Some preliminary code is needed for the `\tl_range:nnn` family of functions.

`\tl_range:Nnn` To avoid checking for the end of the token list at every step, start by counting the number *l* of items and “normalizing” the bounds, namely clamping them to the interval $[0, l]$ and dealing with negative indices. More precisely, `__tl_range_items:nnNn` receives the number of items to skip at the beginning of the token list, the index of the last item to keep, a function among `__tl_range:w`, `__tl_range_braced:w`, `__tl_range_unbraced:w`, and the token list itself. If nothing should be kept, leave `{}`: this

`\tl_range_unbraced:Nnn`
`\tl_range_unbraced:cnn`
`\tl_range_unbraced:nnn`

```

\__tl_range:Nnnn
\__tl_range:nnnNn
\__tl_range:nnNn
\__tl_range_skip:w
\__tl_range_braced:w
\__tl_range_collect_braced:w
\__tl_range_unbraced:w

```

stops the `f`-expansion of `\tl_head:f` and that function produces an empty result. Otherwise, repeatedly call `__tl_range_skip:w` to delete `#1` items from the input stream (the extra brace group avoids an off-by-one shift). For the braced version `__tl_range_braced:w` sets up `__tl_range_collect_braced:w` which stores items one by one in an argument after the semicolon. The unbraced version is almost identical. The version preserving braces and spaces starts by deleting spaces before the argument to avoid collecting them, and sets up `__tl_range_collect:nn` with a first argument of the form `{\langle collected \rangle \langle tokens \rangle}`, whose head is the collected tokens and whose tail is what remains of the original token list. This form makes it easier to move tokens to the `\langle collected \rangle` tokens. Depending on the first token of the tail, either just move it (if it is a space) or also decrement the number of items left to find. Eventually, the result is a brace group followed by the rest of the token list, and `\tl_head:f` cleans up and gives the result in `\exp_not:n`.

```

25498 \cs_new:Npn \tl_range:Nnn { \exp_args:No \tl_range:nnn }
25499 \cs_generate_variant:Nn \tl_range:Nnn { c }
25500 \cs_new:Npn \tl_range:nnn { \__tl_range:Nnnn \__tl_range:w }
25501 \cs_new:Npn \tl_range_braced:Nnn { \exp_args:No \tl_range_braced:nnn }
25502 \cs_generate_variant:Nn \tl_range_braced:Nnn { c }
25503 \cs_new:Npn \tl_range_braced:nnn { \__tl_range:Nnnn \__tl_range_braced:w }
25504 \cs_new:Npn \tl_range_unbraced:Nnn
25505   { \exp_args:No \tl_range_unbraced:nnn }
25506 \cs_generate_variant:Nn \tl_range_unbraced:Nnn { c }
25507 \cs_new:Npn \tl_range_unbraced:nnn
25508   { \__tl_range:Nnnn \__tl_range_unbraced:w }
25509 \cs_new:Npn \__tl_range:Nnnn #1#2#3#4
25510   {
25511     \tl_head:f
25512     {
25513       \exp_args:Nf \__tl_range:nnnNn
25514       { \tl_count:n {#2} } {#3} {#4} #1 {#2}
25515     }
25516   }
25517 \cs_new:Npn \__tl_range:nnnNn #1#2#3
25518   {
25519     \exp_args:Nff \__tl_range:nnNn
25520     {
25521       \exp_args:Nf \__tl_range_normalize:nn
25522       { \int_eval:n { #2 - 1 } } {#1}
25523     }
25524     {
25525       \exp_args:Nf \__tl_range_normalize:nn
25526       { \int_eval:n {#3} } {#1}
25527     }
25528   }
25529 \cs_new:Npn \__tl_range:nnNn #1#2#3#4
25530   {
25531     \if_int_compare:w #2 > #1 \exp_stop_f: \else:
25532       \exp_after:wN { \exp_after:wN }
25533     \fi:
25534     \exp_after:wN #3
25535     \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
25536     \exp_after:wN { \exp:w \__tl_range_skip:w #1 ; { } #4 }
25537   }

```



```

25538 \cs_new:Npn \__tl_range_skip:w #1 ; #2
25539 {
25540   \if_int_compare:w #1 > 0 \exp_stop_f:
25541     \exp_after:wN \__tl_range_skip:w
25542     \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
25543   \else:
25544     \exp_after:wN \exp_end:
25545   \fi:
25546 }
25547 \cs_new:Npn \__tl_range_braced:w #1 ; #2
25548 { \__tl_range_collect_braced:w #1 ; { } #2 }
25549 \cs_new:Npn \__tl_range_unbraced:w #1 ; #2
25550 { \__tl_range_collect_unbraced:w #1 ; { } #2 }
25551 \cs_new:Npn \__tl_range_collect_braced:w #1 ; #2#3
25552 {
25553   \if_int_compare:w #1 > 1 \exp_stop_f:
25554     \exp_after:wN \__tl_range_collect_braced:w
25555     \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
25556   \fi:
25557   { #2 {#3} }
25558 }
25559 \cs_new:Npn \__tl_range_collect_unbraced:w #1 ; #2#3
25560 {
25561   \if_int_compare:w #1 > 1 \exp_stop_f:
25562     \exp_after:wN \__tl_range_collect_unbraced:w
25563     \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
25564   \fi:
25565   { #2 #3 }
25566 }
25567 \cs_new:Npn \__tl_range:w #1 ; #2
25568 {
25569   \exp_args:Nf \__tl_range_collect:nn
25570   { \__tl_range_skip_spaces:n {#2} } {#1}
25571 }
25572 \cs_new:Npn \__tl_range_skip_spaces:n #1
25573 {
25574   \tl_if_head_is_space:nTF {#1}
25575   { \exp_args:Nf \__tl_range_skip_spaces:n {#1} }
25576   { { } #1 }
25577 }
25578 \cs_new:Npn \__tl_range_collect:nn #1#2
25579 {
25580   \int_compare:nNnTF {#2} = 0
25581   {#1}
25582   {
25583     \exp_args:No \tl_if_head_is_space:nTF { \use_none:n #1 }
25584     {
25585       \exp_args:Nf \__tl_range_collect:nn
25586       { \__tl_range_collect_space:nw #1 }
25587       {#2}
25588     }
25589     {
25590       \__tl_range_collect:ff
25591       {

```

```

25592         \exp_args:No \tl_if_head_is_N_type:nTF { \use_none:n #1 }
25593         { \__tl_range_collect_N:nN }
25594         { \__tl_range_collect_group:nn }
25595         #1
25596     }
25597     { \int_eval:n { #2 - 1 } }
25598 }
25599 }
25600 }
25601 \cs_new:Npn \__tl_range_collect_space:nw #1 ~ { { #1 ~ } }
25602 \cs_new:Npn \__tl_range_collect_N:nN #1#2 { { #1 #2 } }
25603 \cs_new:Npn \__tl_range_collect_group:nn #1#2 { { #1 {#2} } }
25604 \cs_generate_variant:Nn \__tl_range_collect:nn { ff }

```

(End definition for `\tl_range:Nnn` and others. These functions are documented on page 248.)

`__tl_range_normalize:nn` This function converts an *<index>* argument into an explicit position in the token list (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the *<index>* #1 and the string count #2. If #1 is negative, replace it by #1 + #2 + 1, then limit to the range [0, #2].

```

25605 \cs_new:Npn \__tl_range_normalize:nn #1#2
25606 {
25607     \int_eval:n
25608     {
25609         \if_int_compare:w #1 < 0 \exp_stop_f:
25610         \if_int_compare:w #1 < -#2 \exp_stop_f:
25611         0
25612         \else:
25613         #1 + #2 + 1
25614         \fi:
25615     \else:
25616     \if_int_compare:w #1 < #2 \exp_stop_f:
25617     #1
25618     \else:
25619     #2
25620     \fi:
25621     \fi:
25622 }
25623 }

```

(End definition for `__tl_range_normalize:nn`.)

43.14 Additions to `l3token`

`\c_catcode_active_space_tl` While `\char_generate:nn` can produce active characters in some engines it cannot in general. It would be possible to simply change the catcode of space but then the code would need to avoid all spaces, making it quite unreadable. Instead we use the primitive `\tex_lowercase:D` trick.

```

25624 \group_begin:
25625     \char_set_catcode_active:N *
25626     \char_set_lccode:nn { '*' } { '\ }
25627     \tex_lowercase:D { \tl_const:Nn \c_catcode_active_space_tl { * } }
25628 \group_end:

```

(End definition for `\c_catcode_active_space_tl`. This variable is documented on page 250.)

25629 `<@@=peek>`

`\peek_N_type:TF` All tokens are N-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be outer, we cannot use the convenient `\bool_if:nTF` function, and must resort to the old trick of using `\ifodd` to expand a set of tests. The `false` branch of this test is taken if the token is one of the first three kinds of non-N-type tokens (explicit or implicit), thus we call `__peek_false:w`. In the `true` branch, we must detect outer tokens, without impacting performance too much for non-outer tokens. The first filter is to search for `outer` in the `\meaning` of `\l_peek_token`. If that is absent, `\use_none_delimit_by_q_stop:w` cleans up, and we call `__peek_true:w`. Otherwise, the token can be a non-outer macro or a primitive mark whose parameter or replacement text contains `outer`, it can be the primitive `\outer`, or it can be an outer token. Macros and marks would have `ma` in the part before the first occurrence of `outer`; the meaning of `\outer` has nothing after `outer`, contrarily to outer macros; and that covers all cases, calling `__peek_true:w` or `__peek_false:w` as appropriate. Here, there is no *<search token>*, so we feed a dummy `\scan_stop:` to the `__peek_token_generic:NNTF` function.

```

25630 \group_begin:
25631   \cs_set_protected:Npn \__peek_tmp:w #1 \q_stop
25632   {
25633     \cs_new_protected:Npn \__peek_execute_branches_N_type:
25634     {
25635       \if_int_odd:w
25636         \if_catcode:w \exp_not:N \l_peek_token { 0 \exp_stop_f: \fi:
25637         \if_catcode:w \exp_not:N \l_peek_token } 0 \exp_stop_f: \fi:
25638         \if_meaning:w \l_peek_token \c_space_token 0 \exp_stop_f: \fi:
25639         1 \exp_stop_f:
25640         \exp_after:wN \__peek_N_type:w
25641         \token_to_meaning:N \l_peek_token
25642         \q_mark \__peek_N_type_aux:nnw
25643         #1 \q_mark \use_none_delimit_by_q_stop:w
25644         \q_stop
25645         \exp_after:wN \__peek_true:w
25646       \else:
25647         \exp_after:wN \__peek_false:w
25648       \fi:
25649     }
25650     \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \q_mark ##3
25651     { ##3 {##1} {##2} }
25652   }
25653   \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \q_stop
25654 \group_end:
25655 \cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
25656 {
25657   \fi:
25658   \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
25659   { \__peek_true:w }
25660   { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
25661 }
25662 \cs_new_protected:Npn \peek_N_type:TF
25663 {

```

```

25664     \__peek_token_generic:NNTF
25665     \__peek_execute_branches_N_type: \scan_stop:
25666   }
25667   \cs_new_protected:Npn \peek_N_type:T
25668   { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
25669   \cs_new_protected:Npn \peek_N_type:F
25670   { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }

(End definition for \peek_N_type:TF and others. This function is documented on page 251.)

25671 \</initex | package>

```

44 l3drivers Implementation

```

25672 \<*initex | package>
25673 \<@=driver>

```

Whilst there is a reasonable amount of code overlap between drivers, it is much clearer to have the blocks more-or-less separated than run in together and DocStripped out in parts. As such, most of the following is set up on a per-driver basis, though there is some common code (again given in blocks not interspersed with other material).

All the file identifiers are up-front so that they come out in the right place in the files.

```

25674 \<*package>
25675 \ProvidesExplFile
25676 \<*dvipdfmx>
25677   {l3dvidpfmx.def}{2018-04-30}{}
25678   {L3 Experimental driver: dvipdfmx}
25679 \</dvipdfmx>
25680 \<*dvips>
25681   {l3dvips.def}{2018-04-30}{}
25682   {L3 Experimental driver: dvips}
25683 \</dvips>
25684 \<*dvisvgm>
25685   {l3dvisvgm.def}{2018-04-30}{}
25686   {L3 Experimental driver: dvisvgm}
25687 \</dvisvgm>
25688 \<*pdfmode>
25689   {l3pdfmode.def}{2018-04-30}{}
25690   {L3 Experimental driver: PDF mode}
25691 \</pdfmode>
25692 \<*xdvipdfmx>
25693   {l3xdvidpfmx.def}{2018-04-30}{}
25694   {L3 Experimental driver: xdvipdfmx}
25695 \</xdvipdfmx>
25696 \</package>

```

The order of the driver code here is such that we get somewhat logical outcomes in terms of code sharing whilst keeping things readable. (Trying to mix all of the code by concept is almost unmanageable.) The key parts which are shared are

- Color support is either `dvips`-like or `pdfmode`-like.
- `pdfmode` and `(x)dvipdfmx` share drawing routines.
- `xdvipdfmx` is largely the same as `dvipdfmx` so takes most of the same code.

`_driver_literal_x:n` The one shared function for all drivers is access to the basic `\special` primitive: it has slightly odd expansion behaviour so a wrapper is provided.

```

\driver_literal_n:n
\driver_literal_x:n
25697 \cs_new_eq:NN \_driver_literal_x:n \tex_special:D
25698 \cs_new_protected:Npn \_driver_literal:n #1
25699 { \_driver_literal_x:n { \exp_not:n {#1} } }
25700 \cs_generate_variant:Nn \_driver_literal:n { x }

```

(End definition for `_driver_literal_x:n` and `_driver_literal:n`.)

44.1 Color support

Color support is split into two parts: a “general” concept and one directly linked to drawings (or rather the split between filling and stroking). General color is relatively easy to handle: we have a color stack available with all modern drivers, and can use that. Whilst (x)dvipdfmx does have its own approach to color specials, it is easier to use dvips-like ones for all cases except direct PDF output.

44.1.1 dvips-style

```

25701 <*dvisvgm | dvipdfmx | dvips | xdvipdfmx>

```

`\driver_color_pickup:N` Allow for L^AT_EX 2_ε color. Here, the possible input values are limited: dvips-style colors can mainly be taken as-is with the exception spot ones (here we need a model and a tint).

```

25702 <*package>
25703 \cs_new_protected:Npn \driver_color_pickup:N #1 { }
25704 \AtBeginDocument
25705 {
25706   \@ifpackageloaded { color }
25707   {
25708     \cs_set_protected:Npn \driver_color_pickup:N #1
25709     {
25710       \exp_args:NV \tl_if_head_is_space:nTF \current@color
25711       {
25712         \tl_set:Nx #1
25713         {
25714           spot ~
25715           \exp_after:wN \use:n \current@color \c_space_tl 1
25716         }
25717       }
25718       {
25719         \exp_after:wN \_driver_color_pickup:w
25720         \current@color \q_stop #1
25721       }
25722     }
25723     \cs_new_protected:Npn \_driver_color_pickup:w #1 ~ #2 \q_stop #3
25724     { \tl_set:Nn #3 { #1 ~ #2 } }
25725   }
25726 { }
25727 }
25728 </package>

```

(End definition for `\driver_color_pickup:N` and `_driver_color_pickup:w`. This function is documented on page 253.)

`\driver_color_cmyk:nnnn` Push the data to the stack. In the case of dvips also reset the drawing fill color in raw PostScript.

`\driver_color_gray:n`

`\driver_color_rgb:nnn`

`\driver_color_spot:nn`

`_driver_color_select:n`

`_driver_color_select:x`

`_driver_color_reset:`

```

25729 \cs_new_protected:Npn \driver_color_cmyk:nnnn #1#2#3#4
25730 {
25731   \_driver_color_select:x
25732   {
25733     cmyk~
25734     \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
25735     \fp_eval:n {#3} ~ \fp_eval:n {#4}
25736   }
25737 }
25738 \cs_new_protected:Npn \driver_color_gray:n #1
25739 { \_driver_color_select:x { gray~ \fp_eval:n {#1} } }
25740 \cs_new_protected:Npn \driver_color_rgb:nnn #1#2#3
25741 {
25742   \_driver_color_select:x
25743   { rgb~ \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} }
25744 }
25745 \cs_new_protected:Npn \driver_color_spot:nn #1#2
25746 { \_driver_color_select:n { \c_space_tl #1 } }
25747 \cs_new_protected:Npn \_driver_color_select:n #1
25748 {
25749   \_driver_literal:n { color~push~ #1 }
25750 }*dvips
25751 \_driver_literal_postscript:n { /l3fc~{ }~def }
25752 </dvips>
25753 \group_insert_after:N \_driver_color_reset:
25754 }
25755 \cs_generate_variant:Nn \_driver_color_select:n { x }
25756 \cs_new_protected:Npn \_driver_color_reset:
25757 { \_driver_literal:n { color~pop } }

```

(End definition for `\driver_color_cmyk:nnnn` and others. These functions are documented on page 253.)

```

25758 </dvisvgm | dvipdfmx | dvips | xdvipdfmx>

```

44.1.2 pdfmode

`\driver_color_pickup:N` The current color in driver-dependent format: pick up the package-mode data if available.

`_driver_color_pickup:w` We end up converting back and forward in this route as we store our color data in dvips format. The `\current@color` needs to be x-expanded before `_driver_color_pickup:w` breaks it apart, because for instance xcolor sets it to be instructions to generate a colour

```

25760 <*package>
25761 \cs_new_protected:Npn \driver_color_pickup:N #1 { }
25762 \AtBeginDocument
25763 {
25764   \@ifpackageloaded { color }
25765   {
25766     \cs_set_protected:Npn \driver_color_pickup:N #1
25767     {
25768       \exp_last_unbraced:Nx \_driver_color_pickup:w

```

```

25769         { \current@color } ~ 0 ~ 0 ~ 0 \q_stop #1
25770     }
25771 \cs_new_protected:Npn \__driver_color_pickup:w
25772     #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 \q_stop #7
25773     {
25774         \str_if_eq:nnTF {#2} { g }
25775         { \tl_set:Nn #7 { gray ~ #1 } }
25776         {
25777             \str_if_eq:nnTF {#4} { rg }
25778             { \tl_set:Nn #7 { rgb ~ #1 ~ #2 ~ #3 } }
25779             {
25780                 \str_if_eq:nnTF {#5} { k }
25781                 { \tl_set:Nn #7 { cmyk ~ #1 ~ #2 ~ #3 ~ #4 } }
25782                 {
25783                     \str_if_eq:nnTF {#2} { cs }
25784                     {
25785                         \tl_set:Nx #7 { spot ~ \use_none:n #1 ~ #5 }
25786                     }
25787                     {
25788                         \tl_set:Nn #7 { gray ~ 0 }
25789                     }
25790                 }
25791             }
25792         }
25793     }
25794 }
25795 { }
25796 }
25797 \end{package}

```

(End definition for `\driver_color_pickup:N` and `__driver_color_pickup:w`. This function is documented on page 253.)

`\l__driver_color_stack_int` pdfTeX and LuaTeX have multiple stacks available, and to track which one is in use a variable is required.

```

25798 \int_new:N \l__driver_color_stack_int

```

(End definition for `\l__driver_color_stack_int`.)

Simply dump the data, but allowing for LuaTeX.

```

\driver_color_cmyk:nnnn \cs_new_protected:Npn \driver_color_cmyk:nnnn #1#2#3#4
\__driver_color_cmyk:nnnn {
\driver_color_gray:n \use:x
\__driver_color_gray:n {
\driver_color_rgb:nnn \__driver_color_cmyk:nnnn
\driver_color_spot:nn { \fp_eval:n {#1} }
\__driver_color_select:n { \fp_eval:n {#2} }
\__driver_color_select:x { \fp_eval:n {#3} }
\__driver_color_reset: { \fp_eval:n {#4} }
}
\cs_new_protected:Npn \__driver_color_cmyk:nnnn #1#2#3#4
{
\__driver_color_select:n
{ #1 ~ #2 ~ #3 ~ #4 ~ k ~ #1 ~ #2 ~ #3 ~ #4 ~ K }
}

```

```

25814 }
25815 \cs_new_protected:Npn \driver_color_gray:n #1
25816 { \exp_args:Nx \__driver_color_gray:n { \fp_eval:n {#1} } }
25817 \cs_new_protected:Npn \__driver_color_gray:n #1
25818 { \__driver_color_select:n { #1 ~ g ~ #1 ~ G } }
25819 \cs_new_protected:Npn \driver_color_rgb:nnn #1#2#3
25820 {
25821   \use:x
25822   {
25823     \__driver_color_rgb:nnn
25824     { \fp_eval:n {#1} }
25825     { \fp_eval:n {#2} }
25826     { \fp_eval:n {#3} }
25827   }
25828 }
25829 \cs_new_protected:Npn \__driver_color_rgb:nnn #1#2#3
25830 { \__driver_color_select:n { #1 ~ #2 ~ #3 ~ rg ~ #1 ~ #2 ~ #3 ~ RG } }
25831 \cs_new_protected:Npn \driver_color_spot:nn #1#2
25832 { \__driver_color_select:n { /#1 ~ cs ~ /#1 ~ CS ~ #2 ~ sc ~ #2 ~ SC } }
25833 \cs_new_protected:Npx \__driver_color_select:n #1
25834 {
25835   \cs_if_exist:NTF \luatex_pdfextension:D
25836   { \luatex_pdfextension:D colorstack }
25837   { \pdfTEX_pdfcolorstack:D }
25838   \exp_not:N \l__driver_color_stack_int push {#1}
25839   \group_insert_after:N \exp_not:N \__driver_color_reset:
25840 }
25841 \cs_generate_variant:Nn \__driver_color_select:n { x }
25842 \cs_new_protected:Npx \__driver_color_reset:
25843 {
25844   \cs_if_exist:NTF \luatex_pdfextension:D
25845   { \luatex_pdfextension:D colorstack }
25846   { \pdfTEX_pdfcolorstack:D }
25847   \exp_not:N \l__driver_color_stack_int pop \scan_stop:
25848 }

```

(End definition for \driver_color_cmyk:nnnn and others. These functions are documented on page 253.)

```
25849 </pdfmode>
```

44.2 dvips driver

```
25850 <*dvips>
```

44.2.1 Basics

_driver_literal_postscript:n Literal PostScript can be included using a few low-level formats. Here, we use the form with no positioning: this is overall more convenient as a wrapper. Note that this does require that where position is important, an appropriate wrapper is included.

```

25851 \cs_new_protected:Npn \__driver_literal_postscript:n #1
25852 { \__driver_literal:n { ps:: #1 } }
25853 \cs_generate_variant:Nn \__driver_literal_postscript:n { x }

```

(End definition for __driver_literal_postscript:n.)

`_driver_align_currentpoint_begin:` In **dvips** there is no build-in saving of the current position, and so some additional
`_driver_align_currentpoint_end:` PostScript is required to set up the transformation matrix and also to restore it afterwards. Notice the use of the stack to save the current position “up front” and to move back to it at the end of the process. Notice that the `[begin]/[end]` pair here mean that we can use a run of PostScript statements in separate lines: not *required* but does make the code and output more clear.

```

25854 \cs_new_protected:Npn \_driver_align_currentpoint_begin:
25855 {
25856   \_driver_literal:n { ps:[begin] }
25857   \_driver_literal_postscript:n { currentpoint }
25858   \_driver_literal_postscript:n { currentpoint~translate }
25859 }
25860 \cs_new_protected:Npn \_driver_align_currentpoint_end:
25861 {
25862   \_driver_literal_postscript:n { neg-exch-neg-exch~translate }
25863   \_driver_literal:n { ps:[end] }
25864 }
```

(End definition for `_driver_align_currentpoint_begin:` and `_driver_align_currentpoint_end:.`)

`_driver_scope_begin:` Saving/restoring scope for general operations needs to be done with **dvips** positioning
`_driver_scope_end:` (try without to see this!). Thus we need the **ps:** version of the special here. As only the graphics state is ever altered within this pairing, we use the lower-cost **g**-versions.

```

25865 \cs_new_protected:Npn \_driver_scope_begin:
25866 { \_driver_literal:n { ps:gsave } }
25867 \cs_new_protected:Npn \_driver_scope_end:
25868 { \_driver_literal:n { ps:grestore } }
```

(End definition for `_driver_scope_begin:` and `_driver_scope_end:.`)

44.2.2 Box operations

`\driver_box_use_clip:N` The **dvips** driver scales all absolute dimensions based on the output resolution selected and any T_EX magnification. Thus for any operation involving absolute lengths there is a correction to make. See `normalscale` from `special.pro` for the variables, noting that here everything is saved on the stack rather than as a separate variable. Once all of that is done, the actual clipping is trivial.

```

25869 \cs_new_protected:Npn \driver_box_use_clip:N #1
25870 {
25871   \_driver_scope_begin:
25872   \_driver_align_currentpoint_begin:
25873   \_driver_literal_postscript:n { matrix~currentmatrix }
25874   \_driver_literal_postscript:n
25875     { Resolution~72~div~VResolution~72~div~scale }
25876   \_driver_literal_postscript:n { DVImag~dup~scale }
25877   \_driver_literal_postscript:x
25878     {
25879       0 ~
25880       \dim_to_decimal_in_bp:n { \box_dp:N #1 } ~
25881       \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
25882       \dim_to_decimal_in_bp:n { -\box_ht:N #1 - \box_dp:N #1 } ~
25883       rectclip
25884     }
```

```

25885   \_driver_literal_postscript:n { setmatrix }
25886   \_driver_align_currentpoint_end:
25887   \hbox_overlap_right:n { \box_use:N #1 }
25888   \_driver_scope_end:
25889   \skip_horizontal:n { \box_wd:N #1 }
25890 }

```

(End definition for \driver_box_use_clip:N. This function is documented on page 252.)

\driver_box_use_rotate:Nn

Rotating using dvips does not require that the box dimensions are altered and has a very convenient built-in operation. Zero rotation must be written as 0 not -0 so there is a quick test.

_driver_box_use_rotate:Nn

```

25891 \cs_new_protected:Npn \driver_box_use_rotate:Nn #1#2
25892 { \exp_args:NNf \_driver_box_use_rotate:Nn #1 { \fp_eval:n {#2} } }
25893 \cs_new_protected:Npn \_driver_box_use_rotate:Nn #1#2
25894 {
25895   \_driver_scope_begin:
25896   \_driver_align_currentpoint_begin:
25897   \_driver_literal_postscript:x
25898   {
25899     \fp_compare:nNnTF {#2} = \c_zero_fp
25900     { 0 }
25901     { \fp_eval:n { round ( -(#2) , 5 ) } } ~
25902     rotate
25903   }
25904   \_driver_align_currentpoint_end:
25905   \box_use:N #1
25906   \_driver_scope_end:
25907 }

```

(End definition for \driver_box_use_rotate:Nn and _driver_box_use_rotate:Nn. This function is documented on page 253.)

\driver_box_use_scale:Nnn

The dvips driver once again has a dedicated operation we can use here.

```

25908 \cs_new_protected:Npn \driver_box_use_scale:Nnn #1#2#3
25909 {
25910   \_driver_scope_begin:
25911   \_driver_align_currentpoint_begin:
25912   \_driver_literal_postscript:x
25913   {
25914     \fp_eval:n { round ( #2 , 5 ) } ~
25915     \fp_eval:n { round ( #3 , 5 ) } ~
25916     scale
25917   }
25918   \_driver_align_currentpoint_end:
25919   \hbox_overlap_right:n { \box_use:N #1 }
25920   \_driver_scope_end:
25921 }

```

(End definition for \driver_box_use_scale:Nnn. This function is documented on page 253.)

44.3 Images

`_driver_image_getbb_eps:n` Simply use the generic function.

```
25922 \cs_new_eq:NN \_driver_image_getbb_eps:n \image_read_bb:n
```

(End definition for `_driver_image_getbb_eps:n`.)

`_driver_image_include_eps:n` The special syntax is relatively clear here: remember we need PostScript sizes here.

```
25923 \cs_new_protected:Npn \_driver_image_include_eps:n #1
```

```
25924 { \_driver_literal:n { PSfile = #1 } }
```

(End definition for `_driver_image_include_eps:n`.)

44.4 Drawing

`_driver_draw_literal:n` The same as literal PostScript: same arguments about positioning apply her.

`_driver_draw_literal:x`

```
25925 \cs_new_eq:NN \_driver_draw_literal:n \_driver_literal_postscript:n
```

```
25926 \cs_generate_variant:Nn \_driver_draw_literal:n { x }
```

(End definition for `_driver_draw_literal:n`.)

`\driver_draw_begin:` The `ps::[begin]` special here deals with positioning but allows us to continue on to a matching `ps::[end]`: contrast with `ps:`, which positions but where we can't split material between separate calls. The `@beginspecial/@endspecial` pair are from `special.pro` and correct the scale and y -axis direction. The definition of `/l3fc` deals with fill color in paths. In contrast to `pgf`, we don't save the current point: discussion with Tom Rokici suggested a better way to handle the necessary translations (see `\driver_draw_box_use:Nnnnn`). (Note that `@beginspecial/@endspecial` forms a driver scope.) The `[begin]/[end]` lines are handled differently from the rest as they are conceptually different: not really drawing literals but instructions to `dvips` itself.

```
25927 \cs_new_protected:Npn \driver_draw_begin:
```

```
25928 {
```

```
25929   \_driver_literal:n { ps::[begin] }
```

```
25930   \_driver_draw_literal:n { @beginspecial }
```

```
25931   \_driver_draw_literal:n { /l3fc-{ }~def }
```

```
25932 }
```

```
25933 \cs_new_protected:Npn \driver_draw_end:
```

```
25934 {
```

```
25935   \_driver_draw_literal:n { @endspecial }
```

```
25936   \_driver_literal:n { ps::[end] }
```

```
25937 }
```

(End definition for `\driver_draw_begin:` and `\driver_draw_end:`. These functions are documented on page 254.)

`\driver_draw_scope_begin:` Scope here may need to contain saved definitions, so the entire memory rather than just the graphic state has to be sent to the stack.

`\driver_draw_scope_end:`

```
25938 \cs_new_protected:Npn \driver_draw_scope_begin:
```

```
25939 { \_driver_draw_literal:n { save } }
```

```
25940 \cs_new_protected:Npn \driver_draw_scope_end:
```

```
25941 { \_driver_draw_literal:n { restore } }
```

(End definition for `\driver_draw_scope_begin:` and `\driver_draw_scope_end:`. These functions are documented on page 254.)

`\driver_draw_moveto:nn` Path creation operations mainly resolve directly to PostScript primitive steps, with only the need to convert to bp. Notice that x-type expansion is included here to ensure that any variable values are forced to literals before any possible caching. There is no native rectangular path command (without also clipping, filling or stroking), so that task is done using a small amount of PostScript.

```

25942 \cs_new_protected:Npn \driver_draw_moveto:nn #1#2
25943 {
25944   \__driver_draw_literal:x
25945   {
25946     \dim_to_decimal_in_bp:n {#1} ~
25947     \dim_to_decimal_in_bp:n {#2} ~ moveto
25948   }
25949 }
25950 \cs_new_protected:Npn \driver_draw_lineto:nn #1#2
25951 {
25952   \__driver_draw_literal:x
25953   {
25954     \dim_to_decimal_in_bp:n {#1} ~
25955     \dim_to_decimal_in_bp:n {#2} ~ lineto
25956   }
25957 }
25958 \cs_new_protected:Npn \driver_draw_rectangle:nnnn #1#2#3#4
25959 {
25960   \__driver_draw_literal:x
25961   {
25962     \dim_to_decimal_in_bp:n {#4} ~ \dim_to_decimal_in_bp:n {#3} ~
25963     \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
25964     moveto~dup~0~rlineto~exch~0~exch~rlineto~neg~0~rlineto~closepath
25965   }
25966 }
25967 \cs_new_protected:Npn \driver_draw_curveto:nnnnnn #1#2#3#4#5#6
25968 {
25969   \__driver_draw_literal:x
25970   {
25971     \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
25972     \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
25973     \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
25974     curveto
25975   }
25976 }

```

(End definition for `\driver_draw_moveto:nn` and others. These functions are documented on page 254.)

`\driver_draw_evenodd_rule:` The even-odd rule here can be implemented as a simply switch.

```

\driver_draw_nonzero_rule:
\g__driver_draw_eor_bool
25977 \cs_new_protected:Npn \driver_draw_evenodd_rule:
25978 { \bool_gset_true:N \g__driver_draw_eor_bool }
25979 \cs_new_protected:Npn \driver_draw_nonzero_rule:
25980 { \bool_gset_false:N \g__driver_draw_eor_bool }
25981 \bool_new:N \g__driver_draw_eor_bool

```

(End definition for `\driver_draw_evenodd_rule:`, `\driver_draw_nonzero_rule:`, and `\g__driver_draw_eor_bool`. These functions are documented on page 255.)

`\driver_draw_closepath:` Unlike PDF, PostScript doesn't track separate colors for strokes and other elements. It is also desirable to have the clip keyword after a stroke or fill. To achieve those outcomes,

```

\driver_draw_stroke:
\driver_draw_closestroke:
\driver_draw_fill:
\driver_draw_fillstroke:
\driver_draw_clip:
\driver_draw_discardpath:
\g__driver_draw_clip_bool

```

there is some work to do. For color, the stroke color is simple but the fill one has to be inserted by hand. For clipping, the required ordering is achieved using a T_EX switch. All of the operations end with a new path instruction as they do not terminate (again in contrast to PDF).

```

25982 \cs_new_protected:Npn \driver_draw_closepath:
25983 { \__driver_draw_literal:n { closepath } }
25984 \cs_new_protected:Npn \driver_draw_stroke:
25985 {
25986   \__driver_draw_literal:n { stroke }
25987   \bool_if:NT \g__driver_draw_clip_bool
25988   {
25989     \__driver_draw_literal:x
25990     {
25991       \bool_if:NT \g__driver_draw_eor_bool { eo }
25992       clip
25993     }
25994   }
25995   \__driver_draw_literal:n { newpath }
25996   \bool_gset_false:N \g__driver_draw_clip_bool
25997 }
25998 \cs_new_protected:Npn \driver_draw_closestroke:
25999 {
26000   \driver_draw_closepath:
26001   \driver_draw_stroke:
26002 }
26003 \cs_new_protected:Npn \driver_draw_fill:
26004 {
26005   \__driver_draw_literal:n { gsave }
26006   \__driver_draw_literal:n { l3fc }
26007   \__driver_draw_literal:x
26008   {
26009     \bool_if:NT \g__driver_draw_eor_bool { eo }
26010     fill
26011   }
26012   \__driver_draw_literal:n { grestore }
26013   \bool_if:NT \g__driver_draw_clip_bool
26014   {
26015     \__driver_draw_literal:x
26016     {
26017       \bool_if:NT \g__driver_draw_eor_bool { eo }
26018       clip
26019     }
26020   }
26021   \__driver_draw_literal:n { newpath }
26022   \bool_gset_false:N \g__driver_draw_clip_bool
26023 }
26024 \cs_new_protected:Npn \driver_draw_fillstroke:
26025 {
26026   \__driver_draw_literal:n { gsave }
26027   \__driver_draw_literal:n { l3fc }
26028   \__driver_draw_literal:x
26029   {
26030     \bool_if:NT \g__driver_draw_eor_bool { eo }
26031     fill

```

```

26032     }
26033     \__driver_draw_literal:n { grestore }
26034     \__driver_draw_literal:n { stroke }
26035     \bool_if:NT \g__driver_draw_clip_bool
26036     {
26037         \__driver_draw_literal:x
26038         {
26039             \bool_if:NT \g__driver_draw_eor_bool { eo }
26040             clip
26041         }
26042     }
26043     \__driver_draw_literal:n { newpath }
26044     \bool_gset_false:N \g__driver_draw_clip_bool
26045 }
26046 \cs_new_protected:Npn \driver_draw_clip:
26047 { \bool_gset_true:N \g__driver_draw_clip_bool }
26048 \bool_new:N \g__driver_draw_clip_bool
26049 \cs_new_protected:Npn \driver_draw_discardpath:
26050 {
26051     \bool_if:NT \g__driver_draw_clip_bool
26052     {
26053         \__driver_draw_literal:x
26054         {
26055             \bool_if:NT \g__driver_draw_eor_bool { eo }
26056             clip
26057         }
26058     }
26059     \__driver_draw_literal:n { newpath }
26060     \bool_gset_false:N \g__driver_draw_clip_bool
26061 }

```

(End definition for \driver_draw_closepath: and others. These functions are documented on page 254.)

\driver_draw_dash_pattern:nn Converting paths to output is again a case of mapping directly to PostScript operations.

```

\__driver_draw_dash:n 26062 \cs_new_protected:Npn \driver_draw_dash_pattern:nn #1#2
\driver_draw_linewidth:n 26063 {
\driver_draw_miterlimit:n 26064     \__driver_draw_literal:x
\driver_draw_cap_but: 26065     {
\driver_draw_cap_round: 26066         [
\driver_draw_cap_rectangle: 26067             \exp_args:Nf \use:n
\driver_draw_join_miter: 26068             { \clist_map_function:nN {#1} \__driver_draw_dash:n }
\driver_draw_join_round: 26069         ] ~
\driver_draw_join_bevel: 26070         \dim_to_decimal_in_bp:n {#2} ~ setdash
26071     }
26072 }
26073 \cs_new:Npn \__driver_draw_dash:n #1
26074 { ~ \dim_to_decimal_in_bp:n {#1} }
26075 \cs_new_protected:Npn \driver_draw_linewidth:n #1
26076 {
26077     \__driver_draw_literal:x
26078     { \dim_to_decimal_in_bp:n {#1} ~ setlinewidth }
26079 }
26080 \cs_new_protected:Npn \driver_draw_miterlimit:n #1

```

```

26081 { \_driver\_draw\_literal:x { \fp\_eval:n {#1} ~ setmiterlimit } }
26082 \cs\_new\_protected:Npn \driver\_draw\_cap\_butt:
26083 { \_driver\_draw\_literal:n { 0 ~ setlinecap } }
26084 \cs\_new\_protected:Npn \driver\_draw\_cap\_round:
26085 { \_driver\_draw\_literal:n { 1 ~ setlinecap } }
26086 \cs\_new\_protected:Npn \driver\_draw\_cap\_rectangle:
26087 { \_driver\_draw\_literal:n { 2 ~ setlinecap } }
26088 \cs\_new\_protected:Npn \driver\_draw\_join\_miter:
26089 { \_driver\_draw\_literal:n { 0 ~ setlinejoin } }
26090 \cs\_new\_protected:Npn \driver\_draw\_join\_round:
26091 { \_driver\_draw\_literal:n { 1 ~ setlinejoin } }
26092 \cs\_new\_protected:Npn \driver\_draw\_join\_bevel:
26093 { \_driver\_draw\_literal:n { 2 ~ setlinejoin } }

```

(End definition for \driver_draw_dash_pattern:nn and others. These functions are documented on page 256.)

\driver_draw_color_fill_cmk:nnnn
\driver_draw_color_stroke_cmk:nnnn
\driver_draw_color_fill_gray:n
\driver_draw_color_stroke_gray:n
\driver_draw_color_fill_rgb:nnn
\driver_draw_color_stroke_rgb:nnn

For dvips, we can use the standard color stack to deal with stroke color, but for fills have to switch to raw PostScript. This is thus not handled by the stack, but the context is very restricted. See also how fills are implemented.

```

26094 \cs\_new\_protected:Npn \driver\_draw\_color\_fill\_cmk:nnnn #1#2#3#4
26095 {
26096   \_driver\_draw\_color\_fill:x
26097   {
26098     \fp\_eval:n {#1} ~ \fp\_eval:n {#2} ~
26099     \fp\_eval:n {#3} ~ \fp\_eval:n {#4} ~
26100     setcmykcolor
26101   }
26102 }
26103 \cs\_new\_protected:Npn \driver\_draw\_color\_stroke\_cmk:nnnn #1#2#3#4
26104 {
26105   \_driver\_draw\_color\_stroke:x
26106   {
26107     cmyk ~
26108     \fp\_eval:n {#1} ~ \fp\_eval:n {#2} ~
26109     \fp\_eval:n {#3} ~ \fp\_eval:n {#4}
26110   }
26111 }
26112 \cs\_new\_protected:Npn \driver\_draw\_color\_fill\_gray:n #1
26113 { \_driver\_draw\_color\_fill:x { \fp\_eval:n {#1} ~ setgray } }
26114 \cs\_new\_protected:Npn \driver\_draw\_color\_stroke\_gray:n #1
26115 { \_driver\_draw\_color\_stroke:x { gray ~ \fp\_eval:n {#1} } }
26116 \cs\_new\_protected:Npn \driver\_draw\_color\_fill\_rgb:nnn #1#2#3
26117 {
26118   \_driver\_draw\_color\_fill:x
26119   { \fp\_eval:n {#1} ~ \fp\_eval:n {#2} ~ \fp\_eval:n {#3} ~ setrgbcolor }
26120 }
26121 \cs\_new\_protected:Npn \driver\_draw\_color\_stroke\_rgb:nnn #1#2#3
26122 {
26123   \_driver\_draw\_color\_stroke:x
26124   { rgb ~ \fp\_eval:n {#1} ~ \fp\_eval:n {#2} ~ \fp\_eval:n {#3} }
26125 }
26126 \cs\_new\_protected:Npn \_driver\_draw\_color\_fill:n #1
26127 { \_driver\_draw\_literal:n { /l3fc ~ { #1 } ~ def } }

```

```

26128 \cs_generate_variant:Nn \__driver_draw_color_fill:n { x }
26129 \cs_new_protected:Npn \__driver_draw_color_stroke:n #1
26130 {
26131   \__driver_literal:n { color-push-#1 }
26132   \group_insert_after:N \__driver_color_reset:
26133 }
26134 \cs_generate_variant:Nn \__driver_draw_color_stroke:n { x }

```

(End definition for `\driver_draw_color_fill_cmyk:nnnn` and others. These functions are documented on page 256.)

`\driver_draw_cm:nnnn` In dvips, keeping the transformations in line with the engine is unfortunately not possible for scaling and rotations: even if we decompose the matrix into those operations, there is still no driver tracking (cf. (x)dvipdfmx). Thus we take the shortest path available and simply dump the matrix as given.

```

26135 \cs_new_protected:Npn \driver_draw_cm:nnnn #1#2#3#4
26136 {
26137   \__driver_draw_literal:n
26138   {
26139     [
26140       \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
26141       \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
26142       0 ~ 0
26143     ] ~
26144     concat
26145   }
26146 }

```

(End definition for `\driver_draw_cm:nnnn`. This function is documented on page 257.)

`\driver_draw_box_use:Nnnnn` Inside a picture `@beginspecial/@endspecial` are active, which is normally a good thing but means that the position and scaling would be off if the box was inserted directly. To deal with that, there are a number of possible approaches. The implementation here was suggested by Tom Rokici (author of dvips). We end the current special placement, then set the current point with a literal `[begin]`. As for general literals, we then use the stack to store the current point and move to it. To insert the required transformation, we have to flip the y -axis, once before and once after it. Then we get back to the TeX reference point to insert our content. The clean up has to happen in the right places, hence the `[begin]/[end]` pair around `restore`. Finally, we can return to “normal” drawing mode. Notice that the set up here is very similar to that in `__driver_align_currentpoint_...`, but the ordering of saving and restoring is different (intermixed).

```

26147 \cs_new_protected:Npn \driver_draw_box_use:Nnnnn #1#2#3#4#5
26148 {
26149   \__driver_draw_literal:n { @endspecial }
26150   \__driver_draw_literal:n { [end] }
26151   \__driver_draw_literal:n { [begin] }
26152   \__driver_draw_literal:n { save }
26153   \__driver_draw_literal:n { currentpoint }
26154   \__driver_draw_literal:n { currentpoint~translate }
26155   \driver_draw_cm:nnnn { 1 } { 0 } { 0 } { -1 }
26156   \driver_draw_cm:nnnn {#2} {#3} {#4} {#5}
26157   \driver_draw_cm:nnnn { 1 } { 0 } { 0 } { -1 }
26158   \__driver_draw_literal:n { neg~exch~neg~exch~translate }

```



```

26159     \_driver_draw_literal:n { [end] }
26160     \hbox_overlap_right:n { \box_use:N #1 }
26161     \_driver_draw_literal:n { [begin] }
26162     \_driver_draw_literal:n { restore }
26163     \_driver_draw_literal:n { [end] }
26164     \_driver_draw_literal:n { [begin] }
26165     \_driver_draw_literal:n { @beginspecial }
26166 }

```

(End definition for \driver_draw_box_use:Nnnnn. This function is documented on page 257.)

```

26167 </dvips>

```

44.5 pdfmode driver

```

26168 <*pdfmode>

```

The direct PDF driver covers both pdfTeX and LuaTeX. The latter renames and restructures the driver primitives but this can be handled at one level of abstraction. As such, we avoid using two separate drivers for this material at the cost of some x-type definitions to get everything expanded up-front.

44.5.1 Basics

_driver_literal_pdf:n This is equivalent to \special{pdf:} but the engine can track it. Without the `direct` keyword everything is kept in sync: the transformation matrix is set to the current point automatically. Note that this is still inside the text (BT ...ET block).

```

26169 \cs_new_protected:Npx \_driver_literal_pdf:n #1
26170 {
26171     \cs_if_exist:NTF \luatex_pdfextension:D
26172     { \luatex_pdfextension:D literal }
26173     { \pdftex_pdfliteral:D }
26174     { \exp_not:N \exp_not:n {#1} }
26175 }
26176 \cs_generate_variant:Nn \_driver_literal_pdf:n { x }

```

(End definition for _driver_literal_pdf:n.)

_driver_scope_begin: Higher-level interfaces for saving and restoring the graphic state.
_driver_scope_end:

```

26177 \cs_new_protected:Npx \_driver_scope_begin:
26178 {
26179     \cs_if_exist:NTF \luatex_pdfextension:D
26180     { \luatex_pdfextension:D save \scan_stop: }
26181     { \pdftex_pdfsave:D }
26182 }
26183 \cs_new_protected:Npx \_driver_scope_end:
26184 {
26185     \cs_if_exist:NTF \luatex_pdfextension:D
26186     { \luatex_pdfextension:D restore \scan_stop: }
26187     { \pdftex_pdfrestore:D }
26188 }

```

(End definition for _driver_scope_begin: and _driver_scope_end:.)

`__driver_matrix:n` Here the appropriate function is set up to insert an affine matrix into the PDF. With
`__driver_matrix:x` pdfTeX and LuaTeX in direct PDF output mode there is a primitive for this, which only
needs the rotation/scaling/skew part.

```

26189 \cs_new_protected:Npx \__driver_matrix:n #1
26190 {
26191   \cs_if_exist:NTF \luatex_pdfextension:D
26192     { \luatex_pdfextension:D setmatrix }
26193     { \pdfsetmatrix:D }
26194     { \exp_not:N \exp_not:n {#1} }
26195 }
26196 \cs_generate_variant:Nn \__driver_matrix:n { x }

```

(End definition for `__driver_matrix:n`.)

44.5.2 Box operations

`\driver_box_use_clip:N` The general method is to save the current location, define a clipping path equivalent to
the bounding box, then insert the content at the current position and in a zero width box.
The “real” width is then made up using a horizontal skip before tidying up. There are
other approaches that can be taken (for example using XForm objects), but the logic here
shares as much code as possible and uses the same conversions (and so same rounding
errors) in all cases.

```

26197 \cs_new_protected:Npn \driver_box_use_clip:N #1
26198 {
26199   \__driver_scope_begin:
26200   \__driver_literal_pdf:x
26201   {
26202     0~
26203     \dim_to_decimal_in_bp:n { -\box_dp:N #1 } ~
26204     \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
26205     \dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
26206     re~W~n
26207   }
26208   \hbox_overlap_right:n { \box_use:N #1 }
26209   \__driver_scope_end:
26210   \skip_horizontal:n { \box_wd:N #1 }
26211 }

```

(End definition for `\driver_box_use_clip:N`. This function is documented on page 252.)

`\driver_box_use_rotate:Nn` Rotations are set using an affine transformation matrix which therefore requires
`__driver_box_use_rotate:Nn` sine/cosine values not the angle itself. We store the rounded values to avoid round-
`\l__driver_cos_fp` ing twice. There are also a couple of comparisons to ensure that -0 is not written to the
`\l__driver_sin_fp` output, as this avoids any issues with problematic display programs. Note that numbers
are compared to 0 after rounding.

```

26212 \cs_new_protected:Npn \driver_box_use_rotate:Nn #1#2
26213 { \exp_args:NNf \__driver_box_use_rotate:Nn #1 { \fp_eval:n {#2} } }
26214 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
26215 {
26216   \__driver_scope_begin:
26217   \box_set_wd:Nn #1 { Opt }
26218   \fp_set:Nn \l__driver_cos_fp { round ( cosd ( #2 ) , 5 ) }
26219   \fp_compare:nNnT \l__driver_cos_fp = \c_zero_fp

```

```

26220     { \fp_zero:N \l__driver_cos_fp }
26221 \fp_set:Nn \l__driver_sin_fp { round ( sind ( #2 ) , 5 ) }
26222 \__driver_matrix:x
26223 {
26224     \fp_use:N \l__driver_cos_fp \c_space_tl
26225     \fp_compare:nNnTF \l__driver_sin_fp = \c_zero_fp
26226     { 0~0 }
26227     {
26228         \fp_use:N \l__driver_sin_fp
26229         \c_space_tl
26230         \fp_eval:n { -\l__driver_sin_fp }
26231     }
26232     \c_space_tl
26233     \fp_use:N \l__driver_cos_fp
26234 }
26235 \box_use:N #1
26236 \__driver_scope_end:
26237 }
26238 \fp_new:N \l__driver_cos_fp
26239 \fp_new:N \l__driver_sin_fp

```

(End definition for `\driver_box_use_rotate:Nn` and others. This function is documented on page 253.)

`\driver_box_use_scale:Nnn` The same idea as for rotation but without the complexity of signs and cosines.

```

26240 \cs_new_protected:Npn \driver_box_use_scale:Nnn #1#2#3
26241 {
26242     \__driver_scope_begin:
26243     \__driver_matrix:x
26244     {
26245         \fp_eval:n { round ( #2 , 5 ) } ~
26246         0~0~
26247         \fp_eval:n { round ( #3 , 5 ) }
26248     }
26249     \hbox_overlap_right:n { \box_use:N #1 }
26250     \__driver_scope_end:
26251 }

```

(End definition for `\driver_box_use_scale:Nnn`. This function is documented on page 253.)

44.6 Images

`\l__driver_image_attr_tl`

In PDF mode, additional attributes of an image (such as page number) are needed both to obtain the bounding box and when inserting the image: this occurs as the image dictionary approach means they are read as part of the bounding box operation. As such, it is easier to track additional attributes using a dedicated `tl` rather than build up the same data twice.

```

26252 \tl_new:N \l__driver_image_attr_tl

```

(End definition for `\l__driver_image_attr_tl`.)

```

\__driver_image_getbb_jpg:n
\__driver_image_getbb_pdf:n
\__driver_image_getbb_png:n
\__driver_image_getbb_auxi:n
\__driver_image_getbb_auxii:n

```

Getting the bounding box here requires us to box up the image and measure it. To deal with the difference in feature support in bitmap and vector images but keeping the common parts, there is a little work to do in terms of auxiliaries. The key here is to

notice that we need two forms of the attributes: a “short” set to allow us to track for caching, and the full form to pass to the primitive.

```

26253 \cs_new_protected:Npn \__driver_image_getbb_jpg:n #1
26254 {
26255   \int_zero:N \l_image_page_int
26256   \tl_clear:N \l_image_pagebox_tl
26257   \tl_set:Nx \l__driver_image_attr_tl
26258     {
26259       \tl_if_empty:NF \l_image_decode_tl
26260       { :D \l_image_decodearray_tl }
26261       \bool_if:NT \l_image_interpolate_bool
26262       { :I }
26263     }
26264   \tl_clear:N \l__driver_image_attr_tl
26265   \__driver_image_getbb_auxi:n {#1}
26266 }
26267 \cs_new_eq:NN \__driver_image_getbb_png:n \__driver_image_getbb_jpg:n
26268 \cs_new_protected:Npn \__driver_image_getbb_pdf:n #1
26269 {
26270   \tl_clear:N \l_image_decode_tl
26271   \bool_set_false:N \l_image_interpolate_bool
26272   \tl_set:Nx \l__driver_image_attr_tl
26273     {
26274       : \l_image_pagebox_tl
26275       \int_compare:nNnT \l_image_page_int > 1
26276       { :P \int_use:N \l_image_page_int }
26277     }
26278   \__driver_image_getbb_auxi:n {#1}
26279 }
26280 \cs_new_protected:Npn \__driver_image_getbb_auxi:n #1
26281 {
26282   \image_bb_restore:xF { #1 \l__driver_image_attr_tl }
26283   { \__driver_image_getbb_auxii:n {#1} }
26284 }
26285 % \begin{macrocode}
26286 % Measuring the image is done by boxing up: for PDF images we could
26287 % use |\pdfTeX pdfximagebbox:D|, but if doesn't work for other types.
26288 % As the box always starts at $(0,0)$ there is no need to worry about
26289 % the lower-left position.
26290 % \begin{macrocode}
26291 \cs_new_protected:Npn \__driver_image_getbb_auxii:n #1
26292 {
26293   \tex_immediate:D \pdfTeX pdfximage:D
26294   \bool_lazy_or:nnT
26295     { \l_image_interpolate_bool }
26296     { ! \tl_if_empty_p:N \l_image_decodearray_tl }
26297     {
26298       attr ~
26299       {
26300         \tl_if_empty:NF \l_image_decode_tl
26301         { /Decode~[ \l_image_decodearray_tl ] }
26302         \bool_if:NT \l_image_interpolate_bool
26303         { /Interpolate~true }
26304       }

```

```

26305     }
26306     \int_compare:nNnT \l_image_page_int > 0
26307     { page ~ \int_use:N \l_image_page_int }
26308     \tl_if_empty:NF \l_image_pagebox_tl
26309     { \l_image_pagebox_tl }
26310     {#1}
26311     \hbox_set:Nn \l__driver_tmp_box
26312     { \pdfrefximage:D \pdfimage:D }
26313     \dim_set:Nn \l_image_urx_dim { \box_wd:N \l__driver_tmp_box }
26314     \dim_set:Nn \l_image_ury_dim { \box_ht:N \l__driver_tmp_box }
26315     \int_const:cn { c__driver_image_ #1 \l__driver_image_attr_tl _int }
26316     { \tex_the:D \pdfimage:D }
26317     \image_bb_save:x { #1 \l__driver_image_attr_tl }
26318 }

```

(End definition for `__driver_image_getbb_jpg:n` and others.)

`__driver_image_include_jpg:n` Images are already loaded for the measurement part of the code, so inclusion is straightforward, with only any attributes to worry about. The latter carry through from determination of the bounding box.

```

26319 \cs_new_protected:Npn \__driver_image_include_jpg:n #1
26320 {
26321     \pdfrefximage:D
26322     \int_use:c { c__driver_image_ #1 \l__driver_image_attr_tl _int }
26323 }
26324 \cs_new_eq:NN \__driver_image_include_pdf:n \__driver_image_include_jpg:n
26325 \cs_new_eq:NN \__driver_image_include_png:n \__driver_image_include_jpg:n

```

(End definition for `__driver_image_include_jpg:n`, `__driver_image_include_pdf:n`, and `__driver_image_include_png:n`.)

```
26326 \pdfmode
```

44.7 dvipdfmx driver

```
26327 (*dvipdfmx | xdvipdfmx)
```

The `dvipdfmx` shares code with the PDF mode one (using the common section to this file) but also with `xdvipdfmx`. The latter is close to identical to `dvipdfmx` and so all of the code here is extracted for both drivers, with some `clean up` for `xdvipdfmx` as required.

44.7.1 Basics

`__driver_literal_pdf:n` Equivalent to `pdf:content` but favored as the link to the pdfTeX primitive approach is clearer.

```

26328 \cs_new_protected:Npn \__driver_literal_pdf:n #1
26329 { \__driver_literal:n { pdf:literal~ #1 } }
26330 \cs_generate_variant:Nn \__driver_literal_pdf:n { x }

```

(End definition for `__driver_literal_pdf:n`.)

`__driver_scope_begin:` Scoping is done using the driver-specific specials.

```

\__driver_scope_end: 26331 \cs_new_protected:Npn \__driver_scope_begin:
26332 { \__driver_literal:n { x:gsave } }
26333 \cs_new_protected:Npn \__driver_scope_end:
26334 { \__driver_literal:n { x:grestore } }

```

(End definition for `__driver_scope_begin:` and `__driver_scope_end:.`)

44.7.2 Box operations

`\driver_box_use_clip:N` The code here is identical to that for `pdfmode`: unlike rotation and scaling, there is no higher-level support in the driver for clipping.

```

26335 \cs_new_protected:Npn \driver_box_use_clip:N #1
26336 {
26337   \__driver_scope_begin:
26338   \__driver_literal_pdf:x
26339   {
26340     0~
26341     \dim_to_decimal_in_bp:n { -\box_dp:N #1 } ~
26342     \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
26343     \dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
26344     re~W~n
26345   }
26346   \hbox_overlap_right:n { \box_use:N #1 }
26347   \__driver_scope_end:
26348   \skip_horizontal:n { \box_wd:N #1 }
26349 }

```

(End definition for `\driver_box_use_clip:N`. This function is documented on page 252.)

`\driver_box_use_rotate:Nn` Rotating in (x)dvipdmfx can be implemented using either PDF or driver-specific code. The former approach however is not “aware” of the content of boxes: this means that any embedded links would not be adjusted by the rotation. As such, the driver-native approach is preferred: the code therefore is similar (though not identical) to the `dvips` version (notice the rotation angle here is positive). As for `dvips`, zero rotation is written as 0 not -0.

```

26350 \cs_new_protected:Npn \driver_box_use_rotate:Nn #1#2
26351 { \exp_args:NNf \__driver_box_use_rotate:Nn #1 { \fp_eval:n {#2} } }
26352 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
26353 {
26354   \__driver_scope_begin:
26355   \__driver_literal:x
26356   {
26357     x:rotate~
26358     \fp_compare:nNnTF {#2} = \c_zero_fp
26359     { 0 }
26360     { \fp_eval:n { round ( #2 , 5 ) } }
26361   }
26362   \box_use:N #1
26363   \__driver_scope_end:
26364 }

```

(End definition for `\driver_box_use_rotate:Nn` and `__driver_box_use_rotate:Nn`. This function is documented on page 253.)

`\driver_box_use_scale:Nnn` Much the same idea for scaling: use the higher-level driver operation to allow for box content.

```

26365 \cs_new_protected:Npn \driver_box_use_scale:Nnn #1#2#3
26366 {
26367   \__driver_scope_begin:
26368   \__driver_literal:x
26369   {

```

```

26370         x:scale~
26371         \fp_eval:n { round ( #2 , 5 ) } ~
26372         \fp_eval:n { round ( #3 , 5 ) }
26373     }
26374     \hbox_overlap_right:n { \box_use:N #1 }
26375     \__driver_scope_end:
26376 }

```

(End definition for \driver_box_use_scale:Nnn. This function is documented on page 253.)

44.8 Images

__driver_image_getbb_eps:n Simply use the generic functions: only for dvipdfmx in the extraction cases.

```

\__driver_image_getbb_jpg:n
\__driver_image_getbb_pdf:n
\__driver_image_getbb_png:n
26377 \cs_new_eq:NN \__driver_image_getbb_eps:n \image_read_bb:n
26378 (*dvipdfmx)
26379 \cs_new_protected:Npn \__driver_image_getbb_jpg:n #1
26380 {
26381     \int_zero:N \l_image_page_int
26382     \tl_clear:N \l_image_pagebox_tl
26383     \image_extract_bb:n {#1}
26384 }
26385 \cs_new_eq:NN \__driver_image_getbb_png:n \__driver_image_getbb_jpg:n
26386 \cs_new_protected:Npn \__driver_image_getbb_pdf:n #1
26387 {
26388     \tl_clear:N \l_image_decode_tl
26389     \bool_set_false:N \l_image_interpolate_bool
26390     \image_extract_bb:n {#1}
26391 }
26392 </dvipdfmx>

```

(End definition for __driver_image_getbb_eps:n and others.)

\g__driver_image_int Used to track the object number associated with each image.

```

26393 \int_new:N \g__driver_image_int

```

(End definition for \g__driver_image_int.)

__driver_image_include_eps:n The special syntax depends on the file type. There is a difference in how PDF images are best handled between dvipdfmx and xdvipdfmx: for the latter it is better to use the primitive route. The relevant code for that is included later in this file.

```

\__driver_image_include_jpg:n
\__driver_image_include_pdf:n
\__driver_image_include_png:n
\__driver_image_include_auxi:nn
\__driver_image_include_auxii:nnn
\__driver_image_include_auxii:xnn
\__driver_image_include_auxiii:nn
26394 \cs_new_protected:Npn \__driver_image_include_eps:n #1
26395 {
26396     \__driver_literal:n { PSfile = #1 }
26397 }
26398 \cs_new_protected:Npn \__driver_image_include_jpg:n #1
26399 { \__driver_image_include_auxi:nn {#1} { image } }
26400 \cs_new_eq:NN \__driver_image_include_png:n \__driver_image_include_jpg:n
26401 (*dvipdfmx)
26402 \cs_new_protected:Npn \__driver_image_include_pdf:n #1
26403 { \__driver_image_include_auxi:nn {#1} { epdf } }
26404 </dvipdfmx>

```

Image inclusion is set up to use the fact that each image is stored in the PDF as an XObject. This means that we can include repeated images only once and refer to them. To allow that, track the nature of each image: much the same as for the direct PDF mode case.

```

26405 \cs_new_protected:Npn \__driver_image_include_auxi:nn #1#2
26406 {
26407   \__driver_image_include_auxii:xnn
26408   {
26409     \tl_if_empty:NF \l_image_pagebox_tl
26410     { : \l_image_pagebox_tl }
26411     \int_compare:nNnT \l_image_page_int > 1
26412     { :P \int_use:N \l_image_page_int }
26413     \tl_if_empty:NF \l_image_decode_tl
26414     { :D \l_image_decodearray_tl }
26415     \bool_if:NT \l_image_interpolate_bool
26416     { :I }
26417   }
26418   {#1} {#2}
26419 }
26420 \cs_new_protected:Npn \__driver_image_include_auxii:nnn #1#2#3
26421 {
26422   \int_if_exist:cTF { c__driver_image_ #2#1 _int }
26423   {
26424     \__driver_literal:x
26425     { pdf:usexobj~@image \int_use:c { c__driver_image_ #2#1 _int } }
26426   }
26427   { \__driver_image_include_auxiii:nn {#2} {#1} {#3} }
26428 }
26429 \cs_generate_variant:Nn \__driver_image_include_auxii:nnn { x }

```

Inclusion using the specials is relatively straight-forward, but there is one wrinkle. To get the pagebox correct for PDF images in all cases, it is necessary to provide both that information and the bbox argument: odd things happen otherwise!

```

26430 \cs_new_protected:Npn \__driver_image_include_auxiii:nnn #1#2#3
26431 {
26432   \int_gincr:N \g__driver_image_int
26433   \int_const:cn { c__driver_image_ #1#2 _int } { \g__driver_image_int }
26434   \__driver_literal:x
26435   {
26436     pdf:#3~
26437     @image \int_use:c { c__driver_image_ #1#2 _int }
26438     \int_compare:nNnT \l_image_page_int > 1
26439     { page ~ \int_use:N \l_image_page_int \c_space_tl }
26440     \tl_if_empty:NF \l_image_pagebox_tl
26441     {
26442       pagebox ~ \l_image_pagebox_tl \c_space_tl
26443       bbox ~
26444       \dim_to_decimal_in_bp:n \l_image_llx_dim \c_space_tl
26445       \dim_to_decimal_in_bp:n \l_image_lly_dim \c_space_tl
26446       \dim_to_decimal_in_bp:n \l_image_urx_dim \c_space_tl
26447       \dim_to_decimal_in_bp:n \l_image_ury_dim \c_space_tl
26448     }
26449     (#1)
26450     \bool_lazy_or:nnT

```



```

26451         { \l_image_interpolate_bool }
26452         { ! \tl_if_empty_p:N \l_image_decodearray_tl }
26453         {
26454             <<
26455             \tl_if_empty:NF \l_image_decode_tl
26456             { /Decode~[ \l_image_decodearray_tl ] }
26457             \bool_if:NT \l_image_interpolate_bool
26458             { /Interpolate~true> }
26459             >>
26460         }
26461     }
26462 }

```

(End definition for `_driver_image_include_eps:n` and others.)

```

26463 </dvipdfmx | xdvipdfmx>

```

44.9 xdvipdfmx driver

```

26464 (*xdvipdfmx)

```

44.10 Images

For `xdvipdfmx`, there are two primitives that allow us to obtain the bounding box without needing `extractbb`. The only complexity is passing the various minor variations to a common core process. The \XeTeX primitive omits the text box from the page box specification, so there is also some “trimming” to do here.

```

\_driver_image_getbb_jpg:n
\_driver_image_getbb_pdf:n
\_driver_image_getbb_png:n
    \_driver_image_getbb_auxi:nN
    \_driver_image_getbb_auxii:nnN
    \_driver_image_getbb_auxiii:nNnn
    \_driver_image_getbb_auxiv:nnNnn
    \_driver_image_getbb_auxv:nNnn
    \_driver_image_getbb_auxvi:nnNnn
    \_driver_image_getbb_pagebox:w
26465 \cs_new_protected:Npn \_driver_image_getbb_jpg:n #1
26466 {
26467     \int_zero:N \l_image_page_int
26468     \tl_clear:N \l_image_pagebox_tl
26469     \_driver_image_getbb_auxi:nN {#1} \xetex_picfile:D
26470 }
26471 \cs_new_eq:NN \_driver_image_getbb_png:n \_driver_image_getbb_jpg:n
26472 \cs_new_protected:Npn \_driver_image_getbb_pdf:n #1
26473 {
26474     \tl_clear:N \l_image_decode_tl
26475     \bool_set_false:N \l_image_interpolate_bool
26476     \_driver_image_getbb_auxi:nN {#1} \xetex_pdffile:D
26477 }
26478 \cs_new_protected:Npn \_driver_image_getbb_auxi:nN #1#2
26479 {
26480     \int_compare:nNnTF \l_image_page_int > 1
26481     { \_driver_image_getbb_auxii:VnN \l_image_page_int {#1} #2 }
26482     { \_driver_image_getbb_auxiii:nNnn {#1} #2 }
26483 }
26484 \cs_new_protected:Npn \_driver_image_getbb_auxii:nnN #1#2#3
26485 { \_driver_image_getbb_aux:nNnn {#2} #3 { :P #1 } { page #1 } }
26486 \cs_generate_variant:Nn \_driver_image_getbb_auxii:nnN { V }
26487 \cs_new_protected:Npn \_driver_image_getbb_auxiii:nNnn #1#2#3#4
26488 {
26489     \tl_if_empty:NTF \l_image_pagebox_tl
26490     { \_driver_image_getbb_auxiv:VnNnn \l_image_pagebox_tl }
26491     { \_driver_image_getbb_auxv:nNnn }
26492     {#1} #2 {#3} {#4}

```

```

26493 }
26494 \cs_new_protected:Npn \__driver_image_getbb_auxiv:nnNnn #1#2#3#4#5
26495 {
26496   \use:x
26497   {
26498     \__driver_image_getbb_auxv:nNnn {#2} #3 { : #1 #4 }
26499     { #5 ~ \__driver_image_getbb_pagebox:w #1 }
26500   }
26501 }
26502 \cs_generate_variant:Nn \__driver_image_getbb_auxiv:nnNnn { V }
26503 \cs_new_protected:Npn \__driver_image_getbb_auxv:nNnn #1#2#3#4
26504 {
26505   \image_bb_restore:nF {#1#3}
26506   { \__driver_image_getbb_auxvi:nNnn {#1} #2 {#3} {#4} }
26507 }
26508 \cs_new_protected:Npn \__driver_image_getbb_auxvi:nNnn #1#2#3#4
26509 {
26510   \hbox_set:Nn \l__driver_tmp_box { #2 #1 ~ #4 }
26511   \dim_set:Nn \l_image_utx_dim { \box_wd:N \l__driver_tmp_box }
26512   \dim_set:Nn \l_image_ury_dim { \box_ht:N \l__driver_tmp_box }
26513   \image_bb_save:n {#1#3}
26514 }
26515 \cs_new:Npn \__driver_image_getbb_pagebox:w #1 box {#1}

```

(End definition for __driver_image_getbb_jpg:n and others.)

__driver_image_include_pdf:n For PDF images, properly supporting the `pagebox` concept in X_YTeX is best done using the `\xetex_pdffile:D` primitive. The syntax here is the same as for the image measurement part, although we know at this stage that there must be some valid setting for `\l_image_pagebox_tl`.

```

26516 \cs_new_protected:Npn \__driver_image_include_pdf:n #1
26517 {
26518   \xetex_pdffile:D "#1" ~
26519   \int_compare:nNnT \l_image_page_int > 0
26520   { page~ \int_use:N \l_image_page_int }
26521   \__driver_image_getbb_auxiv:VnNnn \l_image_pagebox_tl
26522 }

```

(End definition for __driver_image_include_pdf:n.)

```
26523 </xdvipdfmx>
```

44.11 Drawing commands: pdfmode and (x)dvipdfmx

Both `pdfmode` and `(x)dvipdfmx` directly produce PDF output and understand a shared set of specials for drawing commands.

```
26524 <{*dvipdfmx | pdfmode | xdvipdfmx}>
```

44.12 Drawing

__driver_draw_literal:n Pass data through using a dedicated interface.

```

\__driver_draw_literal:x
26525 \cs_new_eq:NN \__driver_draw_literal:n \__driver_literal_pdf:n
26526 \cs_generate_variant:Nn \__driver_draw_literal:n { x }

```

(End definition for `_driver_draw_literal:n`.)

`\driver_draw_begin:` No special requirements here, so simply set up a drawing scope.

```
\driver\_draw\_end: 26527 \cs_new_protected:Npn \driver\_draw\_begin:
                    26528 { \driver\_draw\_scope\_begin: }
                    26529 \cs_new_protected:Npn \driver\_draw\_end:
                    26530 { \driver\_draw\_scope\_end: }
```

(End definition for `\driver_draw_begin:` and `\driver_draw_end:`. These functions are documented on page 254.)

`\driver_draw_scope_begin:` Use the driver-level scope mechanisms.

```
\driver\_draw\_scope\_end: 26531 \cs_new_eq:NN \driver\_draw\_scope\_begin: \_driver\_scope\_begin:
                        26532 \cs_new_eq:NN \driver\_draw\_scope\_end: \_driver\_scope\_end:
```

(End definition for `\driver_draw_scope_begin:` and `\driver_draw_scope_end:`. These functions are documented on page 254.)

`\driver_draw_moveto:nn` Path creation operations all resolve directly to PDF primitive steps, with only the need to convert to bp.

```
\driver\_draw\_lineto:nn 26533 \cs_new_protected:Npn \driver\_draw\_moveto:nn #1#2
\driver\_draw\_curveto:nnnnnn 26534 {
\driver\_draw\_rectangle:nnnn 26535   \_driver\_draw\_literal:x
                             26536   { \dim\_to\_decimal\_in\_bp:n {#1} ~ \dim\_to\_decimal\_in\_bp:n {#2} ~ m }
                             26537 }
                             26538 \cs_new_protected:Npn \driver\_draw\_lineto:nn #1#2
                             26539 {
                             26540   \_driver\_draw\_literal:x
                             26541   { \dim\_to\_decimal\_in\_bp:n {#1} ~ \dim\_to\_decimal\_in\_bp:n {#2} ~ 1 }
                             26542 }
                             26543 \cs_new_protected:Npn \driver\_draw\_curveto:nnnnnn #1#2#3#4#5#6
                             26544 {
                             26545   \_driver\_draw\_literal:x
                             26546   {
                             26547     \dim\_to\_decimal\_in\_bp:n {#1} ~ \dim\_to\_decimal\_in\_bp:n {#2} ~
                             26548     \dim\_to\_decimal\_in\_bp:n {#3} ~ \dim\_to\_decimal\_in\_bp:n {#4} ~
                             26549     \dim\_to\_decimal\_in\_bp:n {#5} ~ \dim\_to\_decimal\_in\_bp:n {#6} ~
                             26550     c
                             26551   }
                             26552 }
                             26553 \cs_new_protected:Npn \driver\_draw\_rectangle:nnnn #1#2#3#4
                             26554 {
                             26555   \_driver\_draw\_literal:x
                             26556   {
                             26557     \dim\_to\_decimal\_in\_bp:n {#1} ~ \dim\_to\_decimal\_in\_bp:n {#2} ~
                             26558     \dim\_to\_decimal\_in\_bp:n {#3} ~ \dim\_to\_decimal\_in\_bp:n {#4} ~
                             26559     re
                             26560   }
                             26561 }
```

(End definition for `\driver_draw_moveto:nn` and others. These functions are documented on page 254.)

`\driver_draw_evenodd_rule:` The even-odd rule here can be implemented as a simply switch.

```
\driver_draw_nonzero_rule:
  \g__driver_draw_eor_bool
26562 \cs_new_protected:Npn \driver_draw_evenodd_rule:
26563   { \bool_gset_true:N \g__driver_draw_eor_bool }
26564 \cs_new_protected:Npn \driver_draw_nonzero_rule:
26565   { \bool_gset_false:N \g__driver_draw_eor_bool }
26566 \bool_new:N \g__driver_draw_eor_bool
```

(End definition for `\driver_draw_evenodd_rule:`, `\driver_draw_nonzero_rule:`, and `\g__driver_draw_eor_bool`. These functions are documented on page 255.)

`\driver_draw_closepath:` Converting paths to output is again a case of mapping directly to PDF operations.

```
\driver_draw_stroke:
\driver_draw_closestroke:
  \driver_draw_fill:
\driver_draw_fillstroke:
  \driver_draw_clip:
\driver_draw_discardpath:
26567 \cs_new_protected:Npn \driver_draw_closepath:
26568   { \__driver_draw_literal:n { h } }
26569 \cs_new_protected:Npn \driver_draw_stroke:
26570   { \__driver_draw_literal:n { S } }
26571 \cs_new_protected:Npn \driver_draw_closestroke:
26572   { \__driver_draw_literal:n { s } }
26573 \cs_new_protected:Npn \driver_draw_fill:
26574   {
26575     \__driver_draw_literal:x
26576     { f \bool_if:NT \g__driver_draw_eor_bool * }
26577   }
26578 \cs_new_protected:Npn \driver_draw_fillstroke:
26579   {
26580     \__driver_draw_literal:x
26581     { B \bool_if:NT \g__driver_draw_eor_bool * }
26582   }
26583 \cs_new_protected:Npn \driver_draw_clip:
26584   {
26585     \__driver_draw_literal:x
26586     { W \bool_if:NT \g__driver_draw_eor_bool * }
26587   }
26588 \cs_new_protected:Npn \driver_draw_discardpath:
26589   { \__driver_draw_literal:n { n } }
```

(End definition for `\driver_draw_closepath:` and others. These functions are documented on page 254.)

`\driver_draw_dash_pattern:nn` Converting paths to output is again a case of mapping directly to PDF operations.

```
\__driver_draw_dash:n
\driver_draw_linewidth:n
\driver_draw_miterlimit:n
\driver_draw_cap_butt:
\driver_draw_cap_round:
\driver_draw_cap_rectangle:
\driver_draw_join_miter:
\driver_draw_join_round:
\driver_draw_join_bevel:
26590 \cs_new_protected:Npn \driver_draw_dash_pattern:nn #1#2
26591   {
26592     \__driver_draw_literal:x
26593     {
26594       [
26595         \exp_args:Nf \use:n
26596         { \clist_map_function:nN {#1} \__driver_draw_dash:n }
26597       ] ~
26598       \dim_to_decimal_in_bp:n {#2} ~ d
26599     }
26600   }
26601 \cs_new:Npn \__driver_draw_dash:n #1
26602   { ~ \dim_to_decimal_in_bp:n {#1} }
26603 \cs_new_protected:Npn \driver_draw_linewidth:n #1
26604   {
26605     \__driver_draw_literal:x
```

```

26606     { \dim_to_decimal_in_bp:n {#1} ~ w }
26607   }
26608   \cs_new_protected:Npn \driver_draw_miterlimit:n #1
26609     { \_driver_draw_literal:x { \fp_eval:n {#1} ~ M } }
26610   \cs_new_protected:Npn \driver_draw_cap_buttt:
26611     { \_driver_draw_literal:n { 0 ~ J } }
26612   \cs_new_protected:Npn \driver_draw_cap_round:
26613     { \_driver_draw_literal:n { 1 ~ J } }
26614   \cs_new_protected:Npn \driver_draw_cap_rectangle:
26615     { \_driver_draw_literal:n { 2 ~ J } }
26616   \cs_new_protected:Npn \driver_draw_join_miter:
26617     { \_driver_draw_literal:n { 0 ~ j } }
26618   \cs_new_protected:Npn \driver_draw_join_round:
26619     { \_driver_draw_literal:n { 1 ~ j } }
26620   \cs_new_protected:Npn \driver_draw_join_bevel:
26621     { \_driver_draw_literal:n { 2 ~ j } }

```

(End definition for \driver_draw_dash_pattern:nn and others. These functions are documented on page 256.)

\driver_draw_color_fill_cmyk:nnnn For the stroke color, all engines here can use the color stack to handle the setting. However, that is not the case for fill color: the stack in (x)dvipdfmx only covers one type of color. So we have to use different approaches for the two sets of engines.

```

\driver_draw_color_fill_gray:n
\driver_draw_color_stroke_cmyk:nnnn
\driver_draw_color_fill_gray:n
\driver_draw_color_stroke_gray:n
\driver_draw_color_fill_rgb:nnn
\driver_draw_color_stroke_rgb:nnn
26622   \cs_new_protected:Npn \driver_draw_color_fill_cmyk:nnnn #1#2#3#4
26623     {
26624       \_driver_color_fill_select:x
26625       {
26626         \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
26627         \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
26628         k
26629       }
26630     }
26631   \cs_new_protected:Npn \driver_draw_color_stroke_cmyk:nnnn #1#2#3#4
26632     {
26633       \_driver_color_select:x
26634       {
26635         \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
26636         \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
26637         k
26638       }
26639     }
26640   \cs_new_protected:Npn \driver_draw_color_fill_gray:n #1
26641     { \_driver_color_fill_select:x { \fp_eval:n {#1} ~ g } }
26642   \cs_new_protected:Npn \driver_draw_color_stroke_gray:n #1
26643     { \_driver_color_select:x { \fp_eval:n {#1} ~ G } }
26644   \cs_new_protected:Npn \driver_draw_color_fill_rgb:nnn #1#2#3
26645     {
26646       \_driver_color_fill_select:x
26647       { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ rg }
26648     }
26649   \cs_new_protected:Npn \driver_draw_color_stroke_rgb:nnn #1#2#3
26650     {
26651       \_driver_color_select:x
26652       { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ RG }

```

```

26653 }
26654 \*pdfmode>
26655 \cs_new_eq:NN \__driver_color_fill_select:n \__driver_color_select:n
26656 \</pdfmode>
26657 \*dvipdfmx | xdvipdfmx>
26658 \cs_new_eq:NN \__driver_color_fill_select:n \__driver_draw_literal:n
26659 \</dvipdfmx | xdvipdfmx>
26660 \cs_generate_variant:Nn \__driver_color_fill_select:n { x }

```

(End definition for \driver_draw_color_fill_cmyk:nnnn and others. These functions are documented on page 256.)

\driver_draw_cm:nnnn
 __driver_draw_cm:nnnn

Another split here between pdfmode and (x)dvipdfmx. In the former, we have a direct method to maintain alignment: the driver can use a matrix itself. For (x)dvipdfmx, we can to decompose the matrix into rotations and a scaling, then use those operations as they are handled by the driver. (There is driver support for matrix operations in (x)dvipdfmx, but as a matched pair so not suitable for the “stand alone” transformation set up here.)

```

26661 \cs_new_protected:Npn \driver_draw_cm:nnnn #1#2#3#4
26662 {
26663 \*pdfmode>
26664 \__driver_matrix:x
26665 {
26666 \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
26667 \fp_eval:n {#3} ~ \fp_eval:n {#4}
26668 }
26669 \</pdfmode>
26670 \*dvipdfmx | xdvipdfmx>
26671 \__driver_draw_cm_decompose:nnnnN {#1} {#2} {#3} {#4}
26672 \__driver_draw_cm:nnnn
26673 \</dvipdfmx | xdvipdfmx>
26674 }
26675 \*dvipdfmx | xdvipdfmx>
26676 \cs_new_protected:Npn \__driver_draw_cm:nnnn #1#2#3#4
26677 {
26678 \__driver_literal:x
26679 {
26680 x:rotate~
26681 \fp_compare:nNnTF {#1} = \c_zero_fp
26682 { 0 }
26683 { \fp_eval:n { round ( -#1 , 5 ) } }
26684 }
26685 \__driver_literal:x
26686 {
26687 x:scale~
26688 \fp_eval:n { round ( #2 , 5 ) } ~
26689 \fp_eval:n { round ( #3 , 5 ) }
26690 }
26691 \__driver_literal:x
26692 {
26693 x:rotate~
26694 \fp_compare:nNnTF {#4} = \c_zero_fp
26695 { 0 }
26696 { \fp_eval:n { round ( -#4 , 5 ) } }

```

```

26697     }
26698   }
26699 </dvipdfmx | xdvipdfmx>

```

(End definition for `\driver_draw_cm:nnnn` and `_driver_draw_cm:nnnn`. This function is documented on page 257.)

```

\_driver_draw_cm_decompose:nnnnN
\_driver_draw_cm_decompose_auxi:nnnnN
\_driver_draw_cm_decompose_auxii:nnnnN
\_driver_draw_cm_decompose_auxiii:nnnnN

```

Internally, transformations for drawing are tracked as a matrix. Not all engines provide a way of dealing with this: if we use a raw matrix, the engine loses track of positions (for example for hyperlinks), and this is not desirable. They do, however, allow us to track rotations and scalings. Luckily, we can decompose any (two-dimensional) matrix into two rotations and a single scaling:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} \cos \beta & \sin \beta \\ -\sin \beta & \cos \beta \end{bmatrix} \begin{bmatrix} w_1 & 0 \\ 0 & w_2 \end{bmatrix} \begin{bmatrix} \cos \gamma & \sin \gamma \\ -\sin \gamma & \cos \gamma \end{bmatrix}$$

The parent matrix can be converted to

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} E & H \\ -H & E \end{bmatrix} + \begin{bmatrix} F & G \\ G & -F \end{bmatrix}$$

From these, we can find that

$$\begin{aligned} \frac{w_1 + w_2}{2} &= \sqrt{E^2 + H^2} \\ \frac{w_1 - w_2}{2} &= \sqrt{F^2 + G^2} \\ \gamma - \beta &= \tan^{-1}(G/F) \\ \gamma + \beta &= \tan^{-1}(H/E) \end{aligned}$$

at which point we just have to do various pieces of re-arrangement to get all of the values. (See J. Blinn, *IEEE Comput. Graph. Appl.*, 1996, **16**, 82–88.) There is one wrinkle: the PostScript (and PDF) way of specifying a transformation matrix exchanges where one would normally expect B and C to be.

```

26700 <*dvipdfmx | xdvipdfmx>
26701 \cs_new_protected:Npn \_driver_draw_cm_decompose:nnnnN #1#2#3#4#5
26702 {
26703   \use:x
26704   {
26705     \_driver_draw_cm_decompose_auxi:nnnnN
26706     { \fp_eval:n { (#1 + #4) / 2 } }
26707     { \fp_eval:n { (#1 - #4) / 2 } }
26708     { \fp_eval:n { (#3 + #2) / 2 } }
26709     { \fp_eval:n { (#3 - #2) / 2 } }
26710   }
26711   #5
26712 }
26713 \cs_new_protected:Npn \_driver_draw_cm_decompose_auxi:nnnnN #1#2#3#4#5
26714 {
26715   \use:x
26716   {
26717     \_driver_draw_cm_decompose_auxii:nnnnN
26718     { \fp_eval:n { 2 * sqrt ( #1 * #1 + #4 * #4 ) } }

```

```

26719         { \fp_eval:n { 2 * sqrt ( #2 * #2 + #3 * #3 ) } }
26720         { \fp_eval:n { atand ( #3 , #2 ) } }
26721         { \fp_eval:n { atand ( #4 , #1 ) } }
26722     }
26723     #5
26724 }
26725 \cs_new_protected:Npn \__driver_draw_cm_decompose_auxii:nnnnN #1#2#3#4#5
26726 {
26727     \use:x
26728     {
26729         \__driver_draw_cm_decompose_auxiii:nnnnN
26730         { \fp_eval:n { ( #4 - #3 ) / 2 } }
26731         { \fp_eval:n { ( #1 + #2 ) / 2 } }
26732         { \fp_eval:n { ( #1 - #2 ) / 2 } }
26733         { \fp_eval:n { ( #4 + #3 ) / 2 } }
26734     }
26735     #5
26736 }
26737 \cs_new_protected:Npn \__driver_draw_cm_decompose_auxiii:nnnnN #1#2#3#4#5
26738 {
26739     \fp_compare:nNnTF { abs ( #2 ) } > { abs ( #3 ) }
26740     { #5 {#1} {#2} {#3} {#4} }
26741     { #5 {#1} {#3} {#2} {#4} }
26742 }
26743 \</dvipdfmx> \xdvipdfmx>

```

(End definition for __driver_draw_cm_decompose:nnnnN and others.)

\driver_draw_box_use:Nnnnn Inserting a T_EX box transformed to the requested position and using the current matrix is done using a mixture of T_EX and low-level manipulation. The offset can be handled by T_EX, so only any rotation/skew/scaling component needs to be done using the matrix operation. As this operation can never be cached, the scope is set directly not using the draw version.

```

26744 \cs_new_protected:Npn \driver_draw_box_use:Nnnnn #1#2#3#4#5
26745 {
26746     \__driver_scope_begin:
26747     \*pdfmode>
26748     \driver_draw_cm:nnnn {#2} {#3} {#4} {#5}
26749     \</pdfmode>
26750     \*dvipdfmx> \xdvipdfmx>
26751     \__driver_literal:x
26752     {
26753         pdf:btrans~matrix~
26754         \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
26755         \fp_eval:n {#4} ~ \fp_eval:n {#5} ~
26756         0 ~ 0
26757     }
26758     \</dvipdfmx> \xdvipdfmx>
26759     \hbox_overlap_right:n { \box_use:N #1 }
26760     \*dvipdfmx> \xdvipdfmx>
26761     \__driver_literal:n { pdf:etrans }
26762     \</dvipdfmx> \xdvipdfmx>
26763     \__driver_scope_end:
26764 }

```


(End definition for `\driver_draw_box_use:Nnnnn`. This function is documented on page 257.)

```
26765 </dvipdfmx | pdfmode | xdvipdfmx>
```

44.13 dvisvgm driver

```
26766 < *dvisvgm>
```

44.13.1 Basics

`_driver_literal_svg:n` Unlike the other drivers, the requirements for making SVG files mean that we can't conveniently transform all operations to the current point. That makes life a bit more tricky later as that needs to be accounted for. A new line is added after each call to help to keep the output readable for debugging.

```
26767 \cs_new_protected:Npn \_driver_literal_svg:n #1
26768 { \_driver_literal:n { dvisvgm:raw~ #1 { ?nl } } }
26769 \cs_generate_variant:Nn \_driver_literal_svg:n { x }
```

(End definition for `_driver_literal_svg:n`.)

`_driver_scope_begin:` A scope in SVG terms is slightly different to the other drivers as operations have to be “tied” to these not simply inside them.

`_driver_scope_end:`

```
26770 \cs_new_protected:Npn \_driver_scope_begin:
26771 { \_driver_literal_svg:n { <g> } }
26772 \cs_new_protected:Npn \_driver_scope_end:
26773 { \_driver_literal_svg:n { </g> } }
```

(End definition for `_driver_scope_begin:` and `_driver_scope_end:.`)

44.14 Driver-specific auxiliaries

`_driver_scope_begin:n` In SVG transformations, clips and so on are attached directly to scopes so we need a way or allowing for that. This is rather more useful than `_driver_scope_begin:` as a result. No assumptions are made about the nature of the scoped operation(s).

`_driver_scope_begin:x`

```
26774 \cs_new_protected:Npn \_driver_scope_begin:n #1
26775 { \_driver_literal_svg:n { <g~ #1 > } }
26776 \cs_generate_variant:Nn \_driver_scope_begin:n { x }
```

(End definition for `_driver_scope_begin:n`.)

44.14.1 Box operations

`\driver_box_use_clip:N` Clipping in SVG is more involved than with other drivers. The first issue is that the clipping path must be defined separately from where it is used, so we need to track how many paths have applied. The naming here uses `l3cp` as the namespace with a number following. Rather than use a rectangular operation, we define the path manually as this allows it to have a depth: easier than the alternative approach of shifting content up and down using scopes to allow for the depth of the \TeX box and keep the reference point the same!

`\g__driver_clip_path_int`

```
26777 \cs_new_protected:Npn \driver_box_use_clip:N #1
26778 {
26779   \int_gincr:N \g__driver_clip_path_int
26780   \_driver_literal_svg:x
26781   { < clipPath-id = " l3cp \int_use:N \g__driver_clip_path_int " > }
```

```

26782 \__driver_literal_svg:x
26783 {
26784   <
26785     path ~ d =
26786       "
26787         M ~ 0 ~
26788           \dim_to_decimal:n { -\box_dp:N #1 } ~
26789         L ~ \dim_to_decimal:n { \box_wd:N #1 } ~
26790           \dim_to_decimal:n { -\box_dp:N #1 } ~
26791         L ~ \dim_to_decimal:n { \box_wd:N #1 } ~
26792           \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
26793         L ~ 0 ~
26794           \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
26795         Z
26796       "
26797     />
26798   }
26799 \__driver_literal_svg:n
26800 { < /clipPath > }

```

In general the SVG set up does not try to transform coordinates to the current point. For clipping we need to do that, so have a transformation here to get us to the right place, and a matching one just before the T_EX box is inserted to get things back on track. The clip path needs to come between those two such that if lines up with the current point, as does the T_EX box.

```

26801 \__driver_scope_begin:n
26802 {
26803   transform =
26804     "
26805       translate ( { ?x } , { ?y } ) ~
26806       scale ( 1 , -1 )
26807     "
26808   }
26809 \__driver_scope_begin:x
26810 {
26811   clip-path =
26812     "url ( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int ) "
26813   }
26814 \__driver_scope_begin:n
26815 {
26816   transform =
26817     "
26818       scale ( -1 , 1 ) ~
26819       translate ( { ?x } , { ?y } ) ~
26820       scale ( -1 , -1 )
26821     "
26822   }
26823 \box_use:N #1
26824 \__driver_scope_end:
26825 \__driver_scope_end:
26826 \__driver_scope_end:
26827 % \skip_horizontal:n { \box_wd:N #1 }
26828 }
26829 \int_new:N \g__driver_clip_path_int

```

(End definition for `\driver_box_use_clip:N` and `\g__driver_clip_path_int`. This function is documented on page 252.)

`\driver_box_use_rotate:Nn` Rotation has a dedicated operation which includes a centre-of-rotation optional pair. That can be picked up from the driver syntax, so there is no need to worry about the transformation matrix.

```

26830 \cs_new_protected:Npn \driver_box_use_rotate:Nn #1#2
26831 {
26832   \__driver_scope_begin:x
26833   {
26834     transform =
26835     "
26836       rotate
26837       ( \fp_eval:n { round ( -(#2) , 5 ) } , ~ { ?x } , ~ { ?y } )
26838     "
26839   }
26840   \box_use:N #1
26841   \__driver_scope_end:
26842 }

```

(End definition for `\driver_box_use_rotate:Nn`. This function is documented on page 253.)

`\driver_box_use_scale:Nnn` In contrast to rotation, we have to account for the current position in this case. That is done using a couple of translations in addition to the scaling (which is therefore done backward with a flip).

```

26843 \cs_new_protected:Npn \driver_box_use_scale:Nnn #1#2#3
26844 {
26845   \__driver_scope_begin:x
26846   {
26847     transform =
26848     "
26849       translate ( { ?x } , { ?y } ) ~
26850       scale
26851       (
26852         \fp_eval:n { round ( -#2 , 5 ) } ,
26853         \fp_eval:n { round ( -#3 , 5 ) }
26854       ) ~
26855       translate ( { ?x } , { ?y } ) ~
26856       scale ( -1 )
26857     "
26858   }
26859   \hbox_overlap_right:n { \box_use:N #1 }
26860   \__driver_scope_end:
26861 }

```

(End definition for `\driver_box_use_scale:Nnn`. This function is documented on page 253.)

44.15 Images

`__driver_image_getbb_png:n` These can be included by extracting the bounding box data.

```

\__driver_image_getbb_jpg:n
26862 \cs_new_eq:NN \__driver_image_getbb_png:n \image_extract_bb:n
26863 \cs_new_eq:NN \__driver_image_getbb_jpg:n \image_extract_bb:n

```

(End definition for `__driver_image_getbb_png:n` and `__driver_image_getbb_jpg:n`.)

```

\__driver_image_include_png:n
\__driver_image_include_jpg:n
\__driver_image_include_bitmap_quote:w

```

The driver here has built-in support for basic image inclusion (see `dvisvgm.def` for a more complex approach, needed if clipping, *etc.*, is covered at the image driver level). The only issue is that `#1` must be quote-corrected. The `dvisvgm:img` operation quotes the file name, but if it is already quoted (contains spaces) then we have an issue: we simply strip off any quotes as a result.

```

26864 \cs_new_protected:Npn \__driver_image_include_png:n #1
26865 {
26866     \__driver_literal:x
26867     {
26868         dvisvgm:img~
26869         \dim_to_decimal:n { \l_image_ury_dim } ~
26870         \dim_to_decimal:n { \l_image_ury_dim } ~
26871         \__driver_image_include_bitmap_quote:w #1 " " \q_stop
26872     }
26873 }
26874 \cs_new_eq:NN \__driver_image_include_jpg:n \__driver_image_include_png:n
26875 \cs_new:Npn \__driver_image_include_bitmap_quote:w #1 " #2 " #3 \q_stop
26876 { #1#2 }

```

(End definition for `__driver_image_include_png:n`, `__driver_image_include_jpg:n`, and `__driver_image_include_bitmap_quote:w`.)

44.16 Drawing

```

\__driver_draw_literal:n
\__driver_draw_literal:x

```

The same as the more general literal call.

```

26877 \cs_new_eq:NN \__driver_draw_literal:n \__driver_literal_svg:n
26878 \cs_generate_variant:Nn \__driver_draw_literal:n { x }

```

(End definition for `__driver_draw_literal:n`.)

```

\driver_draw_begin:
\driver_draw_end:

```

A drawing needs to be set up such that the co-ordinate system is translated. That is done inside a scope, which as described below

```

26879 \cs_new_protected:Npn \driver_draw_begin:
26880 {
26881     \driver_draw_scope_begin:
26882     \__driver_draw_scope:n { transform="translate({?x},{?y})~scale(1,-1)" }
26883 }
26884 \cs_new_protected:Npn \driver_draw_end:
26885 { \driver_draw_scope_end: }

```

(End definition for `\driver_draw_begin:` and `\driver_draw_end:`. These functions are documented on page 254.)

```

\driver_draw_scope_begin:
\driver_draw_scope_end:
\__driver_draw_scope:n
\__driver_draw_scope:x
\g_driver_draw_scope_int
\l_driver_draw_scope_int

```

Several settings that with other drivers are “stand alone” have to be given as part of a scope in SVG. As a result, there is a need to provide a mechanism to automatically close these extra scopes. That is done using a dedicated function and a pair of tracking variables. Within each graphics scope we use a global variable to do the work, with a group used to save the value between scopes. The result is that no direct action is needed when creating a scope.

```

26886 \cs_new_protected:Npn \driver_draw_scope_begin:
26887 {
26888     \int_set_eq:NN
26889     \l__driver_draw_scope_int

```

```

26890     \g__driver_draw_scope_int
26891 \group_begin:
26892     \int_gzero:N \g__driver_draw_scope_int
26893 }
26894 \cs_new_protected:Npn \driver_draw_scope_end:
26895 {
26896     \prg_replicate:nn
26897     { \g__driver_draw_scope_int }
26898     { \__driver_draw_literal:n { </g> } }
26899 \group_end:
26900 \int_gset_eq:NN
26901     \g__driver_draw_scope_int
26902     \l__driver_draw_scope_int
26903 }
26904 \cs_new_protected:Npn \__driver_draw_scope:n #1
26905 {
26906     \__driver_draw_literal:n { <g~ #1 > }
26907     \int_gincr:N \g__driver_draw_scope_int
26908 }
26909 \cs_generate_variant:Nn \__driver_draw_scope:n { x }
26910 \int_new:N \g__driver_draw_scope_int
26911 \int_new:N \l__driver_draw_scope_int

```

(End definition for \driver_draw_scope_begin: and others. These functions are documented on page 254.)

\driver_draw_moveto:nn

\driver_draw_lineto:nn

\driver_draw_rectangle:nnnn

\driver_draw_curveto:nnnnnn

__driver_draw_add_to_path:n

\g__driver_draw_path_tl

Once again, some work is needed to get path constructs correct. Rather than write the values as they are given, the entire path needs to be collected up before being output in one go. For that we use a dedicated storage routine, which adds spaces as required. Since paths should be fully expanded there is no need to worry about the internal x-type expansion.

```

26912 \cs_new_protected:Npn \driver_draw_moveto:nn #1#2
26913 {
26914     \__driver_draw_add_to_path:n
26915     { M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }
26916 }
26917 \cs_new_protected:Npn \driver_draw_lineto:nn #1#2
26918 {
26919     \__driver_draw_add_to_path:n
26920     { L ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }
26921 }
26922 \cs_new_protected:Npn \driver_draw_rectangle:nnnn #1#2#3#4
26923 {
26924     \__driver_draw_add_to_path:n
26925     {
26926         M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2}
26927         h ~ \dim_to_decimal:n {#3} ~
26928         v ~ \dim_to_decimal:n {#4} ~
26929         h ~ \dim_to_decimal:n { -#3 } ~
26930         Z
26931     }
26932 }
26933 \cs_new_protected:Npn \driver_draw_curveto:nnnnnn #1#2#3#4#5#6
26934 {

```

```

26935 \__driver_draw_add_to_path:n
26936 {
26937   C ~
26938   \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} ~
26939   \dim_to_decimal:n {#3} ~ \dim_to_decimal:n {#4} ~
26940   \dim_to_decimal:n {#5} ~ \dim_to_decimal:n {#6}
26941 }
26942 }
26943 \cs_new_protected:Npn \__driver_draw_add_to_path:n #1
26944 {
26945   \tl_gset:Nx \g__driver_draw_path_tl
26946   {
26947     \g__driver_draw_path_tl
26948     \tl_if_empty:NF \g__driver_draw_path_tl { \c_space_tl }
26949     #1
26950   }
26951 }
26952 \tl_new:N \g__driver_draw_path_tl

```

(End definition for \driver_draw_moveto:nn and others. These functions are documented on page 254.)

\driver_draw_evenodd_rule: The fill rules here have to be handled as scopes.

```

\driver_draw_nonzero_rule:
26953 \cs_new_protected:Npn \driver_draw_evenodd_rule:
26954 { \__driver_draw_scope:n { fill-rule="evenodd" } }
26955 \cs_new_protected:Npn \driver_draw_nonzero_rule:
26956 { \__driver_draw_scope:n { fill-rule="nonzero" } }

```

(End definition for \driver_draw_evenodd_rule: and \driver_draw_nonzero_rule:. These functions are documented on page 255.)

__driver_draw_path:n Setting fill and stroke effects and doing clipping all has to be done using scopes. This means setting up the various requirements in a shared auxiliary which deals with the bits and pieces. Clipping paths are reused for path drawing: not essential but avoids constructing them twice. Discarding a path needs a separate function as it's not quite the same.

```

\driver_draw_closepath:
\driver_draw_stroke:
\driver_draw_closestroke:
\driver_draw_fill:
\driver_draw_fillstroke:
\driver_draw_clip:
\driver_draw_discardpath:
\g__driver_draw_clip_bool
\g__driver_draw_path_int
26957 \cs_new_protected:Npn \driver_draw_closepath:
26958 { \__driver_draw_add_to_path:n { Z } }
26959 \cs_new_protected:Npn \__driver_draw_path:n #1
26960 {
26961   \bool_if:NTF \g__driver_draw_clip_bool
26962   {
26963     \int_gincr:N \g__driver_clip_path_int
26964     \__driver_draw_literal:x
26965     {
26966       < clipPath-id = " l3cp \int_use:N \g__driver_clip_path_int " >
26967       { ?nl }
26968       <path-d=" \g__driver_draw_path_tl "/> { ?nl }
26969       < /clipPath > { ? nl }
26970       <
26971       use-xlink:href =
26972       "\c_hash_str l3path \int_use:N \g__driver_path_int " ~
26973       #1
26974       />
26975     }

```

```

26976     \_driver_draw_scope:x
26977     {
26978         clip-path =
26979         "url( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int)"
26980     }
26981 }
26982 {
26983     \_driver_draw_literal:x
26984     { <path ~ d=" \g__driver_draw_path_tl " ~ #1 /> }
26985 }
26986 \tl_gclear:N \g__driver_draw_path_tl
26987 \bool_gset_false:N \g__driver_draw_clip_bool
26988 }
26989 \int_new:N \g__driver_path_int
26990 \cs_new_protected:Npn \driver_draw_stroke:
26991 { \_driver_draw_path:n { style="fill:none" } }
26992 \cs_new_protected:Npn \driver_draw_closestroke:
26993 {
26994     \driver_draw_closepath:
26995     \driver_draw_stroke:
26996 }
26997 \cs_new_protected:Npn \driver_draw_fill:
26998 { \_driver_draw_path:n { style="stroke:none" } }
26999 \cs_new_protected:Npn \driver_draw_fillstroke:
27000 { \_driver_draw_path:n { } }
27001 \cs_new_protected:Npn \driver_draw_clip:
27002 { \bool_gset_true:N \g__driver_draw_clip_bool }
27003 \bool_new:N \g__driver_draw_clip_bool
27004 \cs_new_protected:Npn \driver_draw_discardpath:
27005 {
27006     \bool_if:NT \g__driver_draw_clip_bool
27007     {
27008         \int_gincr:N \g__driver_clip_path_int
27009         \_driver_draw_literal:x
27010         {
27011             < clipPath-id = " l3cp \int_use:N \g__driver_clip_path_int " >
27012             { ?nl }
27013             <path-d=" \g__driver_draw_path_tl "/> { ?nl }
27014             < /clipPath >
27015         }
27016         \_driver_draw_scope:x
27017         {
27018             clip-path =
27019             "url( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int)"
27020         }
27021     }
27022     \tl_gclear:N \g__driver_draw_path_tl
27023     \bool_gset_false:N \g__driver_draw_clip_bool
27024 }

```

(End definition for `_driver_draw_path:n` and others. These functions are documented on page 254.)

`\driver_draw_dash_pattern:nn` All of these ideas are properties of scopes in SVG. The only slight complexity is converting the dash array properly (doing any required maths).

```

\__driver_draw_dash:n
\__driver_draw_dash_aux:nn
\driver_draw_linewidth:n
\driver_draw_miterlimit:n
\driver_draw_cap_butt:
\driver_draw_cap_round:
\driver_draw_cap_rectangle:
\driver_draw_join_miter:
\driver_draw_join_round:
\driver_draw_join_bevel:

```

```

27025 \cs_new_protected:Npn \driver_draw_dash_pattern:nn #1#2
27026 {
27027   \use:x
27028   {
27029     \__driver_draw_dash_aux:nn
27030     { \clist_map_function:nn {#1} \__driver_draw_dash:n }
27031     { \dim_to_decimal:n {#2} }
27032   }
27033 }
27034 \cs_new:Npn \__driver_draw_dash:n #1
27035 { , \dim_to_decimal_in_bp:n {#1} }
27036 \cs_new_protected:Npn \__driver_draw_dash_aux:nn #1#2
27037 {
27038   \__driver_draw_scope:x
27039   {
27040     stroke-dasharray =
27041     "
27042       \tl_if_empty:oTF { \use_none:n #1 }
27043       { none }
27044       { \use_none:n #1 }
27045     " ~
27046     stroke-offset=" #2 "
27047   }
27048 }
27049 \cs_new_protected:Npn \driver_draw_linewidth:n #1
27050 { \__driver_draw_scope:x { stroke-width=" \dim_to_decimal:n {#1} " } }
27051 \cs_new_protected:Npn \driver_draw_miterlimit:n #1
27052 { \__driver_draw_scope:x { stroke-miterlimit=" \fp_eval:n {#1} " } }
27053 \cs_new_protected:Npn \driver_draw_cap_but:
27054 { \__driver_draw_scope:n { stroke-linecap="butt" } }
27055 \cs_new_protected:Npn \driver_draw_cap_round:
27056 { \__driver_draw_scope:n { stroke-linecap="round" } }
27057 \cs_new_protected:Npn \driver_draw_cap_rectangle:
27058 { \__driver_draw_scope:n { stroke-linecap="square" } }
27059 \cs_new_protected:Npn \driver_draw_join_miter:
27060 { \__driver_draw_scope:n { stroke-linejoin="miter" } }
27061 \cs_new_protected:Npn \driver_draw_join_round:
27062 { \__driver_draw_scope:n { stroke-linejoin="round" } }
27063 \cs_new_protected:Npn \driver_draw_join_bevel:
27064 { \__driver_draw_scope:n { stroke-linejoin="bevel" } }

```

(End definition for \driver_draw_dash_pattern:nn and others. These functions are documented on page 256.)

\driver_draw_color_fill_cmyk:nnnn SVG fill color has to be covered outside of the stack, as for dvips. Here, we are only allowed RGB colors so there is some conversion to do.

```

\driver_draw_color_fill_gray:n 27065 \cs_new_protected:Npn \driver_draw_color_stroke_cmyk:nnnn #1#2#3#4
\driver_draw_color_stroke_gray:n 27066 {
\driver_draw_color_fill_rgb:nnn 27067   \use:x
\driver_draw_color_stroke_rgb:nnn 27068   {
  \__driver_draw_color_fill:nnn 27069     \fp_eval:n { -100 * ( (#1) * ( 1 - (#4) ) - 1 ) } }
  \fp_eval:n { -100 * ( (#2) * ( 1 - (#4) ) + #4 - 1 ) } }
  \fp_eval:n { -100 * ( (#3) * ( 1 - (#4) ) + #4 - 1 ) } }
  \__driver_draw_color_fill:nnn 27070
  \fp_eval:n { -100 * ( (#1) * ( 1 - (#4) ) - 1 ) } }
  \fp_eval:n { -100 * ( (#2) * ( 1 - (#4) ) + #4 - 1 ) } }
  \fp_eval:n { -100 * ( (#3) * ( 1 - (#4) ) + #4 - 1 ) } }
  \__driver_draw_color_fill:nnn 27071
  \fp_eval:n { -100 * ( (#1) * ( 1 - (#4) ) - 1 ) } }
  \fp_eval:n { -100 * ( (#2) * ( 1 - (#4) ) + #4 - 1 ) } }
  \fp_eval:n { -100 * ( (#3) * ( 1 - (#4) ) + #4 - 1 ) } }
  \__driver_draw_color_fill:nnn 27072

```



```

27073     }
27074   }
27075   \cs_new_eq:NN \driver_draw_color_stroke_cmyk:nnnn \driver_color_cmyk:nnnn
27076   \cs_new_protected:Npn \driver_draw_color_gray:n #1
27077   {
27078     \use:x
27079     {
27080       \__driver_draw_color_gray_aux:n
27081       { \fp_eval:n { 100 * (#3) } }
27082     }
27083   }
27084   \cs_new_protected:Npn \__driver_draw_color_gray_aux:n #1
27085   { \__driver_draw_color_fill:nnn {#1} {#1} {#1} }
27086   \cs_new_eq:NN \driver_draw_color_stroke_gray:n \driver_color_gray:n
27087   \cs_new_protected:Npn \driver_draw_color_rgb:nnn #1#2#3
27088   {
27089     \use:x
27090     {
27091       \__driver_draw_color_fill:nnn
27092       { \fp_eval:n { 100 * (#1) } }
27093       { \fp_eval:n { 100 * (#2) } }
27094       { \fp_eval:n { 100 * (#3) } }
27095     }
27096   }
27097   \cs_new_protected:Npn \__driver_draw_color_fill:nnn #1#2#3
27098   {
27099     \__driver_draw_scope:x
27100     {
27101       fill =
27102       "
27103       rgb
27104       (
27105         #1 \c_percent_str ,
27106         #2 \c_percent_str ,
27107         #3 \c_percent_str
27108       )
27109       "
27110     }
27111   }
27112   \cs_new_eq:NN \driver_draw_color_stroke_rgb:nnn \driver_color_rgb:nnn

```

(End definition for \driver_draw_color_fill_cmyk:nnnn and others. These functions are documented on page 256.)

\driver_draw_cm:nnnn The four arguments here are floats (the affine matrix), the last two are a displacement vector.

```

27113   \cs_new_protected:Npn \driver_draw_cm:nnnn #1#2#3#4
27114   {
27115     \__driver_draw_scope:n
27116     {
27117       transform =
27118       "
27119       matrix
27120       (

```

```

27121         \fp_eval:n {#1} , \fp_eval:n {#2} ,
27122         \fp_eval:n {#3} , \fp_eval:n {#4} ,
27123         Opt , Opt
27124     )
27125 "
27126 }
27127 }

```

(End definition for `\driver_draw_cm:nnnn`. This function is documented on page 257.)

`\driver_draw_box_use:Nnnnn` No special savings can be made here: simply displace the box inside a scope. As there is nothing to re-box, just make the box passed of zero size.

```

27128 \cs_new_protected:Npn \driver_draw_box_use:Nnnnn #1#2#3#4#5#6#7
27129 {
27130     \__driver_scope_begin:
27131     \driver_draw_cm:nnnn {#2} {#3} {#4} {#5}
27132     \__driver_literal_svg:n
27133     {
27134         < g~
27135         stroke="none"~
27136         transform="scale(-1,1)~translate({?x},{?y})~scale(-1,-1)"
27137     >
27138     }
27139     \box_set_wd:Nn #1 { Opt }
27140     \box_set_ht:Nn #1 { Opt }
27141     \box_set_dp:Nn #1 { Opt }
27142     \box_use:N #1
27143     \__driver_literal_svg:n { </g> }
27144     \__driver_scope_end:
27145 }

```

(End definition for `\driver_draw_box_use:Nnnnn`. This function is documented on page 257.)

```

27146 </dvisvgm>
27147 </initex | package>

```

45 l3deprecation implementation

```

27148 <*initex | package>
27149 <@@=deprecation>

```

`_kernel_deprecation_error:Nnn` The `\outer` definition here ensures the command cannot appear in an argument. Use this auxiliary on all commands that have been removed since 2015.

```

27150 \cs_new_protected:Npn \_kernel_deprecation_error:Nnn #1#2#3
27151 {
27152     \etex_protected:D \tex_outer:D \tex_edef:D #1
27153     {
27154         \exp_not:N \_kernel_msg_expandable_error:nnnnn
27155         { kernel } { deprecated-command }
27156         { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
27157         \exp_not:N \_kernel_msg_error:nnxxx
27158         { kernel } { deprecated-command }
27159         { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }

```

```

27160     }
27161 }
27162 \__kernel_deprecation_error:Nnn \file_if_exist_input:nT
27163 { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }
27164 \__kernel_deprecation_error:Nnn \file_if_exist_input:nTF
27165 { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }
27166 \__kernel_deprecation_error:Nnn \c_job_name_tl
27167 { \c_sys_jobname_str } { 2017-01-01 }
27168 \__kernel_deprecation_error:Nnn \dim_case:nnn
27169 { \dim_case:nnF } { 2015-07-14 }
27170 \__kernel_deprecation_error:Nnn \int_case:nnn
27171 { \int_case:nnF } { 2015-07-14 }
27172 \__kernel_deprecation_error:Nnn \int_from_binary:n
27173 { \int_from_bin:n } { 2016-01-05 }
27174 \__kernel_deprecation_error:Nnn \int_from_hexadecimal:n
27175 { \int_from_hex:n } { 2016-01-05 }
27176 \__kernel_deprecation_error:Nnn \int_from_octal:n
27177 { \int_from_oct:n } { 2016-01-05 }
27178 \__kernel_deprecation_error:Nnn \int_to_binary:n
27179 { \int_to_bin:n } { 2016-01-05 }
27180 \__kernel_deprecation_error:Nnn \int_to_hexadecimal:n
27181 { \int_to_hex:n } { 2016-01-05 }
27182 \__kernel_deprecation_error:Nnn \int_to_octal:n
27183 { \int_to_oct:n } { 2016-01-05 }
27184 \__kernel_deprecation_error:Nnn \ior_get_str:NN
27185 { \ior_str_get:NN } { 2018-03-05 }
27186 \__kernel_deprecation_error:Nnn \luatex_if_engine_p:
27187 { \sys_if_engine luatex_p: } { 2017-01-01 }
27188 \__kernel_deprecation_error:Nnn \luatex_if_engine:F
27189 { \sys_if_engine luatex:F } { 2017-01-01 }
27190 \__kernel_deprecation_error:Nnn \luatex_if_engine:T
27191 { \sys_if_engine luatex:T } { 2017-01-01 }
27192 \__kernel_deprecation_error:Nnn \luatex_if_engine:TF
27193 { \sys_if_engine luatex:TF } { 2017-01-01 }
27194 \__kernel_deprecation_error:Nnn \pdfTeX_if_engine_p:
27195 { \sys_if_engine pdftex_p: } { 2017-01-01 }
27196 \__kernel_deprecation_error:Nnn \pdfTeX_if_engine:F
27197 { \sys_if_engine pdftex:F } { 2017-01-01 }
27198 \__kernel_deprecation_error:Nnn \pdfTeX_if_engine:T
27199 { \sys_if_engine pdftex:T } { 2017-01-01 }
27200 \__kernel_deprecation_error:Nnn \pdfTeX_if_engine:TF
27201 { \sys_if_engine pdftex:TF } { 2017-01-01 }
27202 \__kernel_deprecation_error:Nnn \prop_get:cn
27203 { \prop_item:cn } { 2016-01-05 }
27204 \__kernel_deprecation_error:Nnn \prop_get:Nn
27205 { \prop_item:Nn } { 2016-01-05 }
27206 \__kernel_deprecation_error:Nnn \quark_if_recursion_tail_break:N
27207 { } { 2015-07-14 }
27208 \__kernel_deprecation_error:Nnn \quark_if_recursion_tail_break:n
27209 { } { 2015-07-14 }
27210 \__kernel_deprecation_error:Nnn \scan_align_safe_stop:
27211 { protected~commands } { 2017-01-01 }
27212 \__kernel_deprecation_error:Nnn \str_case:nnn
27213 { \str_case:nnF } { 2015-07-14 }

```

```

27214 \__kernel_deprecation_error:Nnn \str_case:onn
27215 { \str_case:onF } { 2015-07-14 }
27216 \__kernel_deprecation_error:Nnn \str_case_x:nnn
27217 { \str_case_x:nnF } { 2015-07-14 }
27218 \__kernel_deprecation_error:Nnn \tl_case:cnn
27219 { \tl_case:cnF } { 2015-07-14 }
27220 \__kernel_deprecation_error:Nnn \tl_case:Nnn
27221 { \tl_case:NnF } { 2015-07-14 }
27222 \__kernel_deprecation_error:Nnn \tl_to_lowercase:n
27223 { \tex_lowercase:D } { 2018-03-05 }
27224 \__kernel_deprecation_error:Nnn \tl_to_uppercase:n
27225 { \tex_uppercase:D } { 2018-03-05 }
27226 \__kernel_deprecation_error:Nnn \xetex_if_engine_p:
27227 { \sys_if_engine_xetex_p: } { 2017-01-01 }
27228 \__kernel_deprecation_error:Nnn \xetex_if_engine:F
27229 { \sys_if_engine_xetex:F } { 2017-01-01 }
27230 \__kernel_deprecation_error:Nnn \xetex_if_engine:T
27231 { \sys_if_engine_xetex:T } { 2017-01-01 }
27232 \__kernel_deprecation_error:Nnn \xetex_if_engine:TF
27233 { \sys_if_engine_xetex:TF } { 2017-01-01 }

```

(End definition for __kernel_deprecation_error:Nnn.)

__cs_generate_variant_loop_warning:nnxxxx

This is left-over from l3expan. It cannot be done there because l3tl is not loaded at that time. Of course what's deprecated is actually some combinations of variants; see l3expan.

```

27234 \__kernel_deprecation_code:nn
27235 {
27236   \cs_set_protected:Npn \__cs_generate_variant_loop_warning:nnxxxx
27237   { \__kernel_msg_error:nnxxxx }
27238 }
27239 {
27240   \cs_set_protected:Npn \__cs_generate_variant_loop_warning:nnxxxx
27241   { \__kernel_msg_warning:nnxxxx }
27242 }

```

(End definition for __cs_generate_variant_loop_warning:nnxxxx.)

27243 </initex | package>

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
!	191
\!	8304
\"	7249, 7252, 25318
\#	6, 4584, 7249, 9424
\\$	4583, 7249, 7252, 19434
\%	4585, 7249, 9426
\&	4576, 7249, 7252, 8304, 8305
&&	190
\'	25318
\(.....	25361
\)	25361
*	191
*	4269, 4292, 7421, 7423, 7427, 7435
**	191
+	190, 191
\,	10645
-	190, 191
\-	285
\.	25318
/	191
\/	284
\:	4582
\::	33, 326, 343, 344, 2463, 2464, <u>2465</u> , 2466, 2467, 2468, 2469, 2471, 2473, 2480, 2483, 2486, 2492, 2623, 2624, 2625, 2626, 2627, 2628, 2629, 2630, 2631, 2632, 2633, 2634, 2635, 2636, 2637, 2638, 2639, 2640, 2641, 2642, 2643, 2644, 2645, 2646, 2647, 2648, 2649, 2650, 2651, 2652, 2653, 2654, 2655, 2656, 2657, 2658, 2659, 2660, 2661, 2662, 2663, 2664, 2666, 2668, 2673, 2678, 2683, 2720, 2721, 2722, 2723, 2726
\::N ..	33, <u>2467</u> , 2635, 2641, 2642, 2643, 2644, 2645, 2657, 2658, 2659, 2723
\::V	33, <u>2486</u> , 2626, 2632, 2643, 2645, 2647, 2649
\::V_unbraced	33, <u>2665</u>
\::c	33, <u>2469</u> , 2624, 2629, 2636, 2641, 2646, 2647, 2648, 2649, 2650, 2662, 2663
\::error	239, 24050
\::f ..	33, <u>2473</u> , 2628, 2631, 2633, 2634, 2641, 2652, 2653, 2655, 2656, 2722
\::f_unbraced	33, <u>2665</u>
\::n	33, <u>2466</u> , 2624, 2625, 2626, 2627, 2628, 2637, 2642, 2643, 2646, 2647, 2650, 2651, 2652, 2653, 2658, 2660, 2661, 2663, 2720, 2723
\::o	33, <u>2471</u> , 2625, 2629, 2630, 2631, 2632, 2633, 2638, 2639, 2642, 2644, 2646, 2648, 2651, 2654, 2655, 2656, 2659, 2661, 2664, 2721
\::o_unbraced	33, <u>2665</u> , 2720, 2721, 2722, 2723
\::p	33, <u>326</u> , <u>2468</u>
\::v	33, <u>2486</u> , 2627
\::v_unbraced	33, <u>2665</u>
\::x	33, <u>2480</u> , 2623, 2635, 2636, 2637, 2638, 2639, 2640, 2657, 2658, 2659, 2660, 2661, 2662, 2663, 2664
\::x_unbraced	33, <u>2665</u> , 2726
<	190
=	190
>	190
?	190
?:	190
\ \ .	1670, 2441, 4578, 7249, 8239, 8257, 8259, 8264, 8265, 8279, 8280, 8283, 8284, 8381, 8471, 8479, 8717, 8725, 8732, 8744, 8745, 8770, 8771, 8778, 8804, 8806, 8807, 8851, 8852, 8865, 8909, 8930, 8931, 8952, 8953, 8954, 8955, 8956, 9003, 9010, 9017, 9429, 9922, 9928, 9935, 11542, 11554, 11560, 12119, 12122, 12123, 12124, 12131, 12134, 12135, 18017, 18020, 18021, 18046, 18047, 18054, 18055, 18514, 19973, 20086, 20087, 20088, 20109, 21439, 21443, 21448, 21482, 21491, 21495, 21500, 21520, 21522, 21523, 21524, 21527, 21529, 21536, 21540, 21543, 21547, 21549, 21553, 21555, 21561, 21563, 21567, 21569, 21573, 21578, 21580, 21622, 21624, 21629, 21631, 21637, 21642, 21643, 21647, 21651, 21661, 21664, 21668, 21669, 21673, 21681, 21744, 23248
\{	4, 4579, 7249, 8302, 8953, 9423, 18761, 21443, 21448, 21495, 21543, 21580, 21669, 21673, 25373
\}	5, 4580, 7249, 8303, 8953, 9425, 21442, 21448,

21544, 21580, 21669, 21673, 25373
 \backslash^{\sim} 7, 10, 103, 194, 195, 196, 197,
 2124, 2747, 3275, 4581, 7249, 7252,
 7254, 7260, 7312, 8318, 9352, 9388,
 15998, 18105, 18178, 18181, 18707,
 18712, 18713, 18714, 18715, 18718,
 18729, 18766, 18829, 18831, 18833,
 18835, 18837, 18839, 19433, 21044,
 21047, 21061, 21064, 21073, 21076,
 21079, 21082, 21096, 21099, 25318
 \wedge 191
 \backslash_{\sim} 4587, 7249, 7252
 \backslash^{\sim} 25318
 \parallel 190
 \backslash^{\sim} 3296, 4586, 7249,
 7252, 9427, 18751, 18755, 18761, 25318

\backslash_{\sqcup} 283, 2028, 4269,
 4292, 7249, 7414, 8302, 8303, 8772,
 8796, 8939, 8945, 8953, 8955, 9046,
 9430, 18222, 18706, 18711, 18755,
 18765, 18949, 21093, 24234, 25626

A

$\backslash A$ 4270, 4293
 $\backslash AA$ 25300
 $\backslash aa$ 25300
 $\backslash above$ 286
 $\backslash abovedisplayshortskip$ 287
 $\backslash abovedisplayskip$ 288
 $\backslash abovewithdelims$ 289
 $\backslash abs$ 191
 $\backslash accent$ 290
 $\backslash acos$ 193
 $\backslash acosd$ 193
 $\backslash acot$ 194
 $\backslash acotd$ 194
 $\backslash acsc$ 193
 $\backslash acscd$ 193
 $\backslash adjdemerits$ 291
 $\backslash adjustspacing$ 990
 $\backslash advance$ 168, 184, 292
 $\backslash AE$ 25301
 $\backslash ae$ 25301
 $\backslash afterassignment$ 293
 $\backslash aftergroup$ 294
 $\backslash alignmark$ 877
 $\backslash aligntab$ 878
 $\backslash asec$ 193
 $\backslash asecd$ 193
 $\backslash asin$ 193
 $\backslash asind$ 193

assert commands:

$\backslash assert_int:n$ 20275, 21257
 $\backslash atan$ 194
 $\backslash atand$ 194
 $\backslash AtBeginDocument$ 543, 9903, 25704, 25762
 $\backslash atop$ 295
 $\backslash atopwithdelims$ 296
 $\backslash attribute$ 879
 $\backslash attributedef$ 880
 $\backslash automaticdiscretionary$ 881
 $\backslash automatichyphenmode$ 883
 $\backslash automatichyphenpenalty$ 885
 $\backslash autospacing$ 1189
 $\backslash autoxspacing$ 1190

B

$\backslash badness$ 297
 $\backslash baselineskip$ 298
 $\backslash batchmode$ 299
 $\backslash begin$ 202,
 206, 14097, 18510, 18512, 26285, 26290

begin internal commands:

$\backslash _regex_begin$ 21180
 $\backslash begincsname$ 887
 $\backslash begingroup$ 13,
 20, 38, 42, 48, 73, 142, 162, 273, 300

$\backslash beginL$ 608
 $\backslash beginR$ 609
 $\backslash belowdisplayshortskip$ 301
 $\backslash belowdisplayskip$ 302
 $\backslash binoppenalty$ 303
 $\backslash bodydir$ 976

bool commands:

$\backslash bool_const:Nn$ 240, 24075
 $\backslash bool_do_until:Nn$ 98, 6401
 $\backslash bool_do_until:nn$ 98, 6407
 $\backslash bool_do_while:Nn$ 98, 6401
 $\backslash bool_do_while:nn$ 98, 6407
 $\backslash bool_gset:N$ 167, 11099
 $\backslash bool_gset:Nn$ 95, 6263
 $\backslash bool_gset_eq:NN$ 95, 6259, 18697, 20578
 $\backslash bool_gset_false:N$ 94,
 6243, 20537, 25980, 25996, 26022,
 26044, 26060, 26565, 26987, 27023
 $\backslash bool_gset_inverse:N$ 167, 11107
 $\backslash bool_gset_true:N$ 94, 6243,
 20588, 25978, 26047, 26563, 27002
 $\backslash bool_if:N\TF$
 .. 95, 237, 2310, 6271, 6396, 6398,
 6402, 6404, 8357, 8362, 8367, 8372,
 8377, 9625, 9632, 10873, 11071,
 11080, 11246, 11271, 11285, 11302,
 11336, 11364, 11383, 11385, 11390,
 11397, 11421, 11468, 19566, 19575,

- 19977, 20055, 20073, 20102, 20222,
20448, 20456, 21711, 21716, 22713,
23130, 25987, 25991, 26009, 26013,
26017, 26030, 26035, 26039, 26051,
26055, 26261, 26302, 26415, 26457,
26576, 26581, 26586, 26961, 27006
\bool_if:nTF 95, 97, 98, 98,
98, 99, 991, 1011, 6285, 6303, 6363,
6369, 6370, 6378, 6384, 6386, 6409,
6418, 6422, 6431, 6529, 20058, 24153
\bool_if_exist:nTF
..... 95, 6299, 10891, 10907
\bool_if_exist_p:N 95, 6299
\bool_if_p:N 95, 6271
\bool_if_p:n 97, 442, 6265,
6268, 6303, 6311, 6388, 6391, 24079
\bool_lazy_all:nTF
..... 96, 97, 97, 6358, 19841
\bool_lazy_all_p:n 97, 6358
\bool_lazy_and:nnTF .. 96, 97, 97,
6367, 6549, 9240, 24977, 25027, 25343
\bool_lazy_and_p:nn 97, 97, 6367
\bool_lazy_any:nTF
..... 96, 97, 97, 6373, 24777
\bool_lazy_any_p:n 97, 97, 6373, 24980
\bool_lazy_or:nnTF 96,
97, 97, 6382, 18111, 23458, 24435,
24645, 24674, 24856, 24885, 25041,
25083, 25119, 25321, 26294, 26450
\bool_lazy_or_p:nn .. 97, 6382, 25346
\bool_log:N 95, 6286
\bool_log:n 95, 6280
\bool_new:N 94,
6241, 6295, 6296, 6297, 6298, 8190,
9383, 10787, 10788, 10791, 10792,
10796, 10891, 10907, 18555, 18984,
20507, 20508, 20510, 20511, 20512,
22420, 25981, 26048, 26566, 27003
\bool_not_p:n 97, 6388
.bool_set:N 167, 11099
\bool_set:Nn 95, 973, 6263
\bool_set_eq:NN 95, 6259, 18691, 20737
\bool_set_false:N 94,
252, 6243, 9469, 9609, 9617, 9627,
9634, 10813, 11256, 11281, 11293,
11313, 11322, 11371, 19541, 19744,
20486, 20550, 20564, 20610, 20678,
22709, 23128, 26271, 26389, 26475
.bool_set_inverse:N 167, 11107
\bool_set_true:N 94, 266, 6243, 9596,
10808, 11254, 11279, 11295, 11311,
11317, 11378, 19546, 19748, 20480,
20676, 20736, 22727, 22742, 22773
\bool_show:N 95, 6286
\bool_show:n 95, 6280
\bool_until_do:Nn 98, 6395
\bool_until_do:nn 98, 6407
\bool_while_do:Nn 98, 6395
\bool_while_do:nn 99, 6407
\bool_xor_p:nn 98, 6389
\c_false_bool
.. 19, 94, 313, 337, 437, 439, 439,
440, 440, 1894, 1934, 1977, 1982,
2022, 2041, 2247, 2254, 2798, 2992,
6241, 6248, 6254, 6349, 6392, 6540,
19216, 19234, 19450, 19486, 19783,
19925, 19942, 19998, 20141, 21359
\g_tmpa_bool 96, 6295
\l_tmpa_bool 95, 6295
\g_tmpb_bool 96, 6295
\l_tmpb_bool 95, 6295
\c_true_bool 19,
94, 313, 357, 437, 439, 439, 440,
440, 1934, 2022, 2040, 2268, 6245,
6251, 6350, 6351, 6393, 6534, 18562,
18694, 19120, 19180, 19230, 19416,
19441, 19485, 19492, 19923, 19933,
19996, 20208, 20386, 20397, 20412,
20554, 20587, 21190, 21267, 21320
bool internal commands:
__bool_!:Nw 6329
__bool_&_0: 6341
__bool_&_1: 6341
__bool_&_2: 6341
__bool_(:Nw 6334
__bool_)_0: 6341
__bool_)_1: 6341
__bool_)_2: 6341
__bool_choose:NNN .. 6336, 6340, 6341
__bool_get_next:NN
. 439, 439, 6317, 6319, 6331, 6337,
6352, 6353, 6354, 6355, 6356, 6357
__bool_lazy_all:n 6358
__bool_lazy_any:n 6373
__bool_p:Nw 6339
__bool_show:NN 6286
__bool_to_str:n 6280, 6293
__bool_|_0: 6341
__bool_|_1: 6341
__bool_|_2: 6341
\botmark 304
\botmarks 610
\box 305
box commands:
__box_autosize_to_wd_and_ht:Nnn ..
..... 222, 22324
__box_autosize_to_wd_and_ht_plus_-
dp:Nnn 222, 22324

- \box_clear:N [215](#),
[215](#), [21770](#), [21777](#), [22455](#), [22515](#), [22562](#)
- \box_clear_new:N [215](#), [21776](#)
- \box_clip:N [236](#), [236](#), [236](#), [23595](#)
- \box_dp:N [217](#),
[12600](#), [21802](#), [21809](#), [22105](#), [22223](#),
[22313](#), [22328](#), [22583](#), [22584](#), [22663](#),
[22668](#), [22686](#), [22700](#), [22863](#), [22884](#),
[23200](#), [23606](#), [23613](#), [23618](#), [23758](#),
[25880](#), [25882](#), [26203](#), [26205](#), [26341](#),
[26343](#), [26788](#), [26790](#), [26792](#), [26794](#)
- \box_gclear:N [215](#), [21770](#), [21779](#)
- \box_gclear_new:N [215](#), [21776](#)
- \box_gset_eq:NN ... [215](#), [21773](#), [21782](#)
- \box_gset_eq_clear:NN ... [215](#), [21790](#)
- \box_gset_to_last:N [218](#), [21844](#)
- \box_ht:N [217](#), [12599](#),
[21802](#), [21811](#), [22104](#), [22222](#), [22312](#),
[22325](#), [22328](#), [22510](#), [22557](#), [22585](#),
[22586](#), [22654](#), [22659](#), [22686](#), [22693](#),
[22862](#), [22883](#), [23198](#), [23623](#), [23631](#),
[23636](#), [23752](#), [23756](#), [25882](#), [26205](#),
[26314](#), [26343](#), [26512](#), [26792](#), [26794](#)
- \box_if_empty:NTF [217](#), [21840](#)
- \box_if_empty_p:N [217](#), [21840](#)
- \box_if_exist:NTF
... [215](#), [21777](#), [21779](#), [21798](#), [21877](#)
- \box_if_exist_p:N [215](#), [21798](#)
- \box_if_horizontal:NTF ... [217](#), [21832](#)
- \box_if_horizontal_p:N ... [217](#), [21832](#)
- \box_if_vertical:NTF [218](#), [21832](#)
- \box_if_vertical_p:N [218](#), [21832](#)
- \box_log:N [219](#), [21863](#)
- \box_log:Nnn [219](#), [21863](#)
- \box_move_down:nn [216](#), [960](#), [21821](#),
[23610](#), [23618](#), [23656](#), [23663](#), [23734](#)
- \box_move_left:nn [216](#), [21821](#)
- \box_move_right:nn [216](#), [21821](#)
- \box_move_up:nn [216](#), [21821](#), [22904](#),
[23195](#), [23627](#), [23636](#), [23670](#), [23683](#)
- \box_new:N [215](#),
[215](#), [21762](#), [21852](#), [21853](#), [21854](#),
[21855](#), [21856](#), [22091](#), [22396](#), [22462](#)
- \box_resize:Nnn [22379](#)
- \box_resize_to_ht:Nn [223](#), [22242](#)
- \box_resize_to_ht_plus_dp:Nn ...
..... [223](#), [22242](#)
- \box_resize_to_wd:Nn [223](#), [22242](#)
- \box_resize_to_wd_and_ht:Nnn ...
..... [223](#), [22242](#)
- \box_resize_to_wd_and_ht_plus_
dp:Nnn [224](#), [22204](#),
[22380](#), [22382](#), [22384](#), [22386](#), [23871](#)
- \box_rotate:Nn [224](#), [22092](#), [23727](#)
- \box_scale:Nnn [224](#), [22300](#), [23895](#)
- \box_set_dp:Nn
.. [217](#), [960](#), [21808](#), [22131](#), [22354](#),
[22357](#), [22863](#), [22884](#), [23199](#), [23613](#),
[23621](#), [23659](#), [23664](#), [23739](#), [27141](#)
- \box_set_eq:NN . [215](#), [21771](#), [21782](#),
[22572](#), [22886](#), [23203](#), [23641](#), [23688](#)
- \box_set_eq_clear:NN [215](#), [21790](#)
- \box_set_ht:Nn
..... [217](#), [21808](#), [22130](#), [22353](#),
[22358](#), [22862](#), [22883](#), [23197](#), [23630](#),
[23639](#), [23673](#), [23686](#), [23737](#), [27140](#)
- \box_set_to_last:N [218](#), [21844](#)
- \box_set_wd:Nn
.. [217](#), [21808](#), [22132](#), [22370](#), [22864](#),
[22885](#), [23201](#), [23740](#), [26217](#), [27139](#)
- \box_show:N [218](#), [21857](#)
- \box_show:Nnn [218](#), [21857](#)
- \box_trim:Nnnnn [236](#), [23598](#)
- \box_use:N
.. [216](#), [216](#), [21817](#), [22119](#), [22901](#),
[22904](#), [23192](#), [23195](#), [23603](#), [23611](#),
[23619](#), [23628](#), [23637](#), [23649](#), [23657](#),
[23663](#), [23671](#), [23684](#), [23735](#), [23742](#),
[25887](#), [25905](#), [25919](#), [26160](#), [26208](#),
[26235](#), [26249](#), [26346](#), [26362](#), [26374](#),
[26759](#), [26823](#), [26840](#), [26859](#), [27142](#)
- \box_use_clear:N [22379](#)
- \box_use_drop:N [216](#),
[21817](#), [22134](#), [22365](#), [22374](#), [22388](#),
[22389](#), [22391](#), [22392](#), [22985](#), [23122](#)
- \box_viewport:Nnnnn [236](#), [23644](#)
- \box_wd:N [217](#),
[12598](#), [21802](#), [21813](#), [22106](#), [22224](#),
[22314](#), [22334](#), [22587](#), [22588](#), [22658](#),
[22667](#), [22675](#), [22680](#), [22827](#), [22864](#),
[22885](#), [22902](#), [23193](#), [23202](#), [23650](#),
[23755](#), [23763](#), [23932](#), [23939](#), [25881](#),
[25889](#), [26204](#), [26210](#), [26313](#), [26342](#),
[26348](#), [26511](#), [26789](#), [26791](#), [26827](#)
- \c_empty_box
.. [215](#), [217](#), [218](#), [21771](#), [21773](#), [21852](#)
- \g_tmpa_box [218](#), [21853](#)
- \l_tmpa_box [218](#), [21853](#)
- \g_tmpb_box [218](#), [21853](#)
- \l_tmpb_box [218](#), [21853](#)
- box internal commands:
- \l_box_angle_fp
.. [22080](#), [22096](#), [22097](#), [22098](#), [22127](#)
- _box_autosize:Nnnn [22324](#)
- \l_box_bottom_dim [22083](#),
[22105](#), [22162](#), [22166](#), [22171](#), [22177](#),
[22182](#), [22186](#), [22195](#), [22197](#), [22214](#),
[22223](#), [22232](#), [22265](#), [22313](#), [22319](#)

\l__box_bottom_new_dim	__box_rotate_y:nnN
22087, 22131, 22163, 22174, 22185,	22092, 22147, 22160, 22162, 22171,
22196, 22231, 22318, 22354, 22358	22173, 22182, 22184, 22193, 22195
\l__box_cos_fp 22081,	__box_scale_aux:N 22300, 22339
22098, 22110, 22115, 22142, 22154	\l__box_scale_x_fp 22202,
__box_dim_eval:n	22209, 22230, 22252, 22267, 22277,
21752, 21809, 21811, 21813, 21822,	22279, 22289, 22304, 22321, 22334,
21824, 21826, 21828, 21912, 21918,	22336, 22337, 22338, 22348, 22360
21948, 21955, 21963, 21985, 22024,	\l__box_scale_y_fp
22030, 22060, 22067, 22079, 23602, 22202, 22211, 22232, 22234,
23604, 23648, 23650, 23659, 23683	22247, 22252, 22262, 22267, 22279,
__box_dim_eval:w 21752	22291, 22305, 22317, 22319, 22335,
\l__box_internal_box 22091, 22119,	22336, 22337, 22338, 22349, 22351
22120, 22126, 22130, 22131, 22132,	__box_show:NNnn . 21861, 21871, 21875
22134, 22344, 22353, 22354, 22357,	\l__box_sin_fp
22358, 22365, 22370, 22374, 23600, 22081, 22097, 22108, 22143, 22153
23608, 23611, 23613, 23616, 23619,	\l__box_top_dim 22083, 22104, 22160,
23621, 23623, 23625, 23628, 23630,	22164, 22173, 22175, 22184, 22188,
23631, 23634, 23636, 23637, 23639,	22193, 22199, 22214, 22222, 22234,
23641, 23646, 23654, 23657, 23659,	22250, 22265, 22294, 22312, 22317
23662, 23663, 23664, 23668, 23671,	\l__box_top_new_dim
23673, 23681, 23684, 23686, 23688	22087, 22130, 22161, 22172, 22183,
\l__box_left_dim 22083, 22107,	22194, 22233, 22316, 22353, 22357
22162, 22164, 22173, 22177, 22182,	\boxdir 977
22188, 22193, 22197, 22225, 22315	\boxmaxdepth 306
\l__box_left_new_dim 22087, 22122,	bp 195
22133, 22165, 22176, 22187, 22198	\breakafterdirmode 888
__box_log:nnN 21863	\brokenpenalty 307
__box_resize:N	
. 22204, 22253, 22268, 22280, 22296	
__box_resize:NNN 22204	
__box_resize_common:N	
. 22235, 22322, 22342	
__box_resize_set_corners:N	
. 22204, 22246, 22261, 22276, 22288	
\l__box_right_dim 22083, 22106,	
22160, 22166, 22171, 22175, 22184,	
22186, 22195, 22199, 22210, 22224,	
22230, 22278, 22290, 22314, 22321	
\l__box_right_new_dim 22087,	
22133, 22167, 22178, 22189, 22200,	
22229, 22320, 22362, 22364, 22370	
__box_rotate:N 22092	
__box_rotate_quadrant_four:	
. 22092, 22191	
__box_rotate_quadrant_one:	
. 22092, 22158	
__box_rotate_quadrant_three:	
. 22092, 22180	
__box_rotate_quadrant_two:	
. 22092, 22169	
__box_rotate_x:nnN	
22092, 22136, 22164, 22166, 22175,	
22177, 22186, 22188, 22197, 22199	
	C
	\c 25318
	\catcode 4, 5, 6, 7, 10, 211, 212, 213, 214,
	215, 216, 217, 218, 219, 224, 225,
	226, 227, 228, 229, 230, 231, 232, 308
	catcode commands:
	\c_catcode_active_space_tl 250, 25624
	\c_catcode_active_tl
 118, 475, 7434, 7494
	\c_catcode_letter_token
 118, 474, 7416, 7484, 18301
	\c_catcode_other_space_tl 114,
	532, 7414, 9397, 9430, 9501, 9589, 9651
	\c_catcode_other_token
 118, 475, 7416, 7489, 18299
	\catcodetable 889
	cc 195
	ceil 192
	\char 309, 7599
	char commands:
	\l_char_active_seq 117, 141, 7247, 9713
	\char_codepoint_to_bytes:n
	250, 24439, 25108, 25128, 25129, 25280
	\char_fold_case:N 250, 24411

- \char_generate:nn . [114](#), [891](#), [1010](#),
[7274](#), [7414](#), [9046](#), [18854](#), [19851](#),
[24433](#), [25047](#), [25090](#), [25102](#), [25104](#),
[25139](#), [25140](#), [25145](#), [25147](#), [25152](#),
[25153](#), [25158](#), [25160](#), [25273](#), [25274](#)
- \char_gset_active_eq:NN [113](#), [7253](#)
- \char_gset_active_eq:nN [113](#), [7253](#)
- \char_lower_case:N [250](#), [24411](#)
- \char_mixed_case:N [250](#), [24411](#)
- \char_set_active_eq:NN [113](#), [7253](#), [9716](#)
- \char_set_active_eq:nN [113](#), [7253](#)
- \char_set_catcode:nn
[116](#), [241](#), [242](#), [243](#), [244](#), [245](#), [246](#),
[247](#), [248](#), [249](#), [7153](#), [7160](#), [7162](#),
[7164](#), [7166](#), [7168](#), [7170](#), [7172](#), [7174](#),
[7176](#), [7178](#), [7180](#), [7182](#), [7184](#), [7186](#),
[7188](#), [7190](#), [7192](#), [7194](#), [7196](#), [7198](#),
[7200](#), [7202](#), [7204](#), [7206](#), [7208](#), [7210](#),
[7212](#), [7214](#), [7216](#), [7218](#), [7220](#), [7222](#)
- \char_set_catcode_active:N
. [115](#), [7159](#), [7254](#),
[7312](#), [7435](#), [8305](#), [18105](#), [21044](#), [25625](#)
- \char_set_catcode_active:n
. [115](#), [7191](#), [7377](#), [10644](#), [10645](#)
- \char_set_catcode_alignment:N
. [115](#), [7159](#), [7423](#), [21096](#)
- \char_set_catcode_alignment:n
. [115](#), [259](#), [7191](#), [7356](#)
- \char_set_catcode_comment:N [115](#), [7159](#)
- \char_set_catcode_comment:n [115](#), [7191](#)
- \char_set_catcode_end_line:N
. [115](#), [7159](#)
- \char_set_catcode_end_line:n
. [115](#), [7191](#)
- \char_set_catcode_escape:N [115](#), [7159](#)
- \char_set_catcode_escape:n [115](#), [7191](#)
- \char_set_catcode_group_begin:N
. [115](#), [7159](#), [18178](#), [21047](#)
- \char_set_catcode_group_begin:n
. [115](#), [7191](#), [7349](#)
- \char_set_catcode_group_end:N
. [115](#), [7159](#), [18181](#), [21064](#)
- \char_set_catcode_group_end:n
. [115](#), [7191](#), [7351](#)
- \char_set_catcode_ignore:N [115](#), [7159](#)
- \char_set_catcode_ignore:n
. [115](#), [256](#), [257](#), [7191](#)
- \char_set_catcode_invalid:N [115](#), [7159](#)
- \char_set_catcode_invalid:n [115](#), [7191](#)
- \char_set_catcode_letter:N
. [115](#), [7159](#), [14238](#), [14239](#), [21073](#)
- \char_set_catcode_letter:n
. [115](#), [260](#), [262](#), [7191](#), [7373](#)
- \char_set_catcode_math_subscript:N
. [115](#), [7159](#), [7427](#), [21061](#)
- \char_set_catcode_math_subscript:n
. [115](#), [7191](#), [7368](#)
- \char_set_catcode_math_superscript:N
. [115](#), [7159](#), [21099](#)
- \char_set_catcode_math_superscript:n
. [115](#), [261](#), [7191](#), [7366](#)
- \char_set_catcode_math_toggle:N
. [115](#), [7159](#), [7421](#), [21076](#)
- \char_set_catcode_math_toggle:n
. [115](#), [7191](#), [7354](#)
- \char_set_catcode_other:N
. [115](#), [7159](#), [18338](#), [21079](#)
- \char_set_catcode_other:n
. [115](#), [258](#), [263](#), [7191](#), [7315](#), [7375](#)
- \char_set_catcode_parameter:N
. [115](#), [7159](#), [21082](#)
- \char_set_catcode_parameter:n
. [115](#), [7191](#), [7364](#)
- \char_set_catcode_space:N . [115](#), [7159](#)
- \char_set_catcode_space:n
. [115](#), [264](#), [7191](#), [7371](#)
- \char_set_lccode:nn
. [116](#), [3281](#), [3298](#), [7223](#), [7260](#),
[7381](#), [7382](#), [8302](#), [8303](#), [8304](#), [25626](#)
- \char_set_mathcode:nn [117](#), [7223](#)
- \char_set_sfcode:nn [117](#), [7223](#)
- \char_set_uccode:nn [116](#), [7223](#)
- \char_show_value_catcode:n [116](#), [7153](#)
- \char_show_value_lccode:n . [116](#), [7223](#)
- \char_show_value_mathcode:n [117](#), [7223](#)
- \char_show_value_sfcode:n . [117](#), [7223](#)
- \char_show_value_uccode:n . [117](#), [7223](#)
- \l_char_special_seq [117](#), [7247](#)
- \char_upper_case:N [250](#), [24411](#)
- \char_value_catcode:n [116](#), [241](#), [242](#),
[243](#), [244](#), [245](#), [246](#), [247](#), [248](#), [249](#),
[3293](#), [3323](#), [7153](#), [24433](#), [25047](#), [25090](#)
- \char_value_lccode:n
. [116](#), [7223](#), [23491](#), [24412](#), [24422](#)
- \char_value_mathcode:n [117](#), [7223](#)
- \char_value_sfcode:n [117](#), [7223](#)
- \char_value_uccode:n [117](#), [7223](#), [24414](#)
- char internal commands:
 _char_change_case:nN [24411](#)
- _char_change_case:nNN [24411](#)
- _char_codepoint_to_bytes_-
 auxi:n [24439](#)
- _char_codepoint_to_bytes_-
 auxii:Nnn [24439](#)
- _char_codepoint_to_bytes_-
 auxiii:n [24439](#)

- _char_data_auxi:w 23463,
23482, 23486, 23518, 23523, 23556
- _char_data_auxii:w
. 23465, 23466, 23500,
23503, 23526, 23527, 23529, 23531
- \g_char_data_ior . . . 23457, 23462,
23479, 23484, 23485, 23513, 23521,
23522, 23545, 23559, 23575, 23576
- _char_generate:w
. 23461, 23474, 23475,
23494, 23495, 23505, 23507, 23508,
23510, 23536, 23538, 23539, 23541
- _char_generate_aux:nn 7274
- _char_generate_aux:nnw 7274
- _char_generate_aux:w . . . 7276, 7280
- _char_generate_auxii:nnw . . . 7274
- _char_generate_invalid_
catcode: 7274
- _char_int_to_roman:w
. 7273, 7386, 7409
- _char_tmp:n
. 7379, 7391, 7394, 7396, 7399
- _char_tmp:NN . . . 23564, 23570, 23572
- _char_tmp:nN 7255, 7266, 7267
- \l_char_tmp_tl 7274
- \chardef 221, 234, 310
- choice commands:
.choice: 167, 11115
- choices commands:
.choices:nn 167, 11117
- \cite 25366
- \cleaders 311
- \clearmarks 890
- clist commands:
\clist_clear:N 105,
105, 6628, 6645, 6808, 11238, 11263
- \clist_clear_new:N 105, 6632
- \clist_concat:NNN 105, 6671, 6697, 6710
- \clist_const:Nn 105, 6625
- \clist_count:N
. 109, 112, 7008, 7037, 7069, 23702
- \clist_count:n 109, 7008, 7100, 23693
- \clist_gclear:N 105, 6628, 6647
- \clist_gclear_new:N 105, 6632
- \clist_gconcat:NNN
. 105, 6671, 6699, 6712
- \clist_get:NN 111, 6722
- \clist_get:NNTF 111, 6759
- \clist_gpop:NN 111, 6733
- \clist_gpop:NNTF 111, 6759
- \clist_gpush:Nn 111, 6784
- \clist_gput_left:Nn
. 106, 6696, 6792, 6793,
6794, 6795, 6796, 6797, 6798, 6799
- \clist_gput_right:Nn 106, 6709
- \clist_gremove_all:Nn 107, 6818
- \clist_gremove_duplicates:N 107, 6802
- \clist_greverse:N 107, 6857
- .clist_gset:N 167, 11127
- \clist_gset:Nn 106, 6690
- \clist_gset_eq:NN . . . 105, 6636, 6805
- \clist_gset_from_seq:NN
. 105, 6644, 6821, 17794
- \clist_gsort:Nn 107, 6875, 17779
- \clist_if_empty:NNTF 107, 6680, 6842,
6875, 6932, 6962, 6982, 11002, 23701
- \clist_if_empty:nTF 108, 6879
- \clist_if_empty_p:N 107, 6875
- \clist_if_empty_p:n 108, 6879
- \clist_if_exist:NNTF
. 105, 6686, 7035, 9804, 9889
- \clist_if_exist_p:N 105, 6686
- \clist_if_in:NnTF 104, 108, 6811, 6893
- \clist_if_in:nnTF . . . 108, 6893, 11992
- \clist_item:Nn
. 112, 464, 962, 7066, 23702
- \clist_item:nn . . . 112, 962, 7097, 23697
- \clist_log:N 112, 7125
- \clist_log:n 112, 7139
- \clist_map_break: 109, 6936, 6941,
6950, 6954, 6970, 6988, 7004, 11436
- \clist_map_break:n
. 109, 6913, 7004, 11379, 17787, 17793
- \clist_map_function:NN 60,
108, 4754, 4764, 6916, 6930, 7013, 7135
- \clist_map_function:Nn 461
- \clist_map_function:nN
. 108, 242, 242, 462, 4759, 4769,
6946, 7144, 11478, 24190, 26068, 26596
- \clist_map_function:nn 27030
- \clist_map_inline:Nn 108, 108, 459,
6809, 6960, 11374, 11427, 17787, 17793
- \clist_map_inline:nn
. 108, 1539, 1547,
3030, 6960, 10963, 11033, 23300, 23313
- \clist_map_variable:NNn . . . 109, 6980
- \clist_map_variable:nNn . . . 109, 6980
- \clist_new:N . . . 104, 105, 449, 6623,
6800, 7146, 7147, 7148, 7149, 10784
- \clist_pop:NN 111, 6733
- \clist_pop:NNTF 111, 6759
- \clist_push:Nn 111, 6784
- \clist_put_left:Nn
. 106, 6696, 6784, 6785,
6786, 6787, 6788, 6789, 6790, 6791
- \clist_put_right:Nn
. 106, 6709, 6812, 11465
- \clist_rand_item:N 236, 23692

- \clist_rand_item:n .. [236](#), [243](#), [23692](#)
- \clist_remove_all:Nn .. [107](#), [6818](#)
- \clist_remove_duplicates:N .. [104](#), [107](#), [6802](#)
- \clist_reverse:N .. [107](#), [6857](#)
- \clist_reverse:n .. [107](#), [457](#), [6858](#), [6860](#), [6863](#)
- .clist_set:N .. [167](#), [11127](#)
- \clist_set:Nn [106](#), [6690](#), [6697](#), [6699](#), [6710](#), [6712](#), [6899](#), [6976](#), [6993](#), [11001](#)
- \clist_set_eq:NN [105](#), [6636](#), [6803](#), [11359](#)
- \clist_set_from_seq:NN .. [105](#), [6644](#), [6819](#), [17788](#)
- \clist_show:N .. [112](#), [112](#), [7125](#)
- \clist_show:n .. [112](#), [112](#), [7139](#)
- \clist_sort:Nn .. [107](#), [6875](#), [17779](#)
- \clist_use:Nn .. [110](#), [7033](#)
- \clist_use:Nnnn .. [110](#), [407](#), [7033](#)
- \c_empty_clist .. [112](#), [6576](#), [6724](#), [6739](#), [6761](#), [6775](#)
- \l_foo_clist .. [200](#)
- \g_tmpa_clist .. [112](#), [7146](#)
- \l_tmpa_clist .. [112](#), [7146](#)
- \g_tmpb_clist .. [112](#), [7146](#)
- \l_tmpb_clist .. [112](#), [7146](#)
- clist internal commands:
 - __clist_concat:NNNN .. [6671](#)
 - __clist_count:n .. [7008](#)
 - __clist_count:w .. [7008](#)
 - __clist_get:wN .. [6722](#), [6764](#)
 - __clist_if_empty_n:w .. [6879](#)
 - __clist_if_empty_n:wNw .. [6879](#)
 - __clist_if_in_return:nnN .. [6893](#)
 - __clist_if_wrap:nTF .. [450](#), [6598](#), [6622](#), [6663](#), [6824](#), [6905](#)
 - __clist_if_wrap:w .. [450](#), [6598](#)
 - \l__clist_internal_clist .. [453](#), [6577](#), [6702](#), [6703](#), [6715](#), [6716](#), [6899](#), [6900](#), [6901](#), [6976](#), [6977](#), [6993](#), [6994](#)
 - \l__clist_internal_remove_clist .. [6800](#), [6808](#), [6811](#), [6812](#), [6814](#)
 - \l__clist_internal_remove_seq .. [6800](#), [6826](#), [6827](#), [6828](#)
 - __clist_item:nnnN .. [7066](#), [7099](#)
 - __clist_item_n:nw .. [7097](#)
 - __clist_item_n_end:n .. [7097](#)
 - __clist_item_N_loop:nw .. [7066](#)
 - __clist_item_n_loop:nw .. [7097](#)
 - __clist_item_n_strip:n .. [7097](#)
 - __clist_item_n_strip:w .. [7097](#)
 - __clist_map_function:Nw .. [459](#), [6930](#), [6967](#)
 - __clist_map_function_n:Nn [460](#), [6946](#)
 - __clist_map_unbrace:Nw .. [460](#), [6946](#)
 - __clist_map_variable:Nnw .. [6980](#)
 - __clist_pop:NNN .. [6733](#)
 - __clist_pop:wN .. [6733](#)
 - __clist_pop:wwNNN .. [454](#), [6733](#), [6778](#)
 - __clist_pop_TF:NNN .. [6759](#)
 - __clist_put_left:NNNn .. [6696](#)
 - __clist_put_right:NNNn .. [6709](#)
 - __clist_rand_item:nn .. [23692](#)
 - __clist_remove_all: .. [6818](#)
 - __clist_remove_all:NNNn .. [6818](#)
 - __clist_remove_all:w .. [456](#), [6818](#)
 - __clist_remove_duplicates:NN .. [6802](#)
 - __clist_reverse:wwNw .. [457](#), [6863](#)
 - __clist_reverse_end:ww .. [457](#), [6863](#)
 - __clist_sanitize:n .. [6585](#), [6626](#), [6691](#), [6693](#)
 - __clist_sanitize:Nn .. [449](#), [6585](#)
 - __clist_set_from_seq:n .. [6644](#)
 - __clist_set_from_seq:NNNN .. [6644](#)
 - __clist_show:NN .. [7125](#)
 - __clist_show:Nn .. [7139](#)
 - __clist_tmp:w .. [456](#), [6578](#), [6831](#), [6853](#), [6907](#), [6916](#), [6920](#), [6922](#)
 - __clist_trim_next:w .. [449](#), [460](#), [6579](#), [6588](#), [6596](#), [6949](#), [6957](#)
 - __clist_use:nwn .. [7033](#)
 - __clist_use:nwwwnwn .. [462](#), [7033](#)
 - __clist_use:wN .. [7033](#)
 - __clist_wrap_item:w .. [449](#), [6594](#), [6621](#)
- \closein .. [312](#)
- \closeout .. [313](#)
- \clubpenalties .. [611](#)
- \clubpenalty .. [314](#)
- cm .. [195](#)
- code commands:
 - .code:n .. [167](#), [11125](#)
- coffin commands:
 - \coffin_attach:NnnNnnnn .. [227](#), [948](#), [22858](#)
 - \coffin_attach_mark:NnnNnnnn .. [22858](#), [23052](#), [23073](#), [23089](#)
 - \coffin_clear:N .. [226](#), [22451](#)
 - \coffin_display_handles:Nn .. [228](#), [23095](#)
 - \coffin_dp:N .. [228](#), [22583](#), [23219](#), [23869](#), [23897](#)
 - \coffin_ht:N .. [228](#), [22583](#), [23218](#), [23869](#), [23897](#)
 - \coffin_if_exist:Ntf .. [226](#), [22430](#), [22444](#)
 - \coffin_if_exist_p:N .. [226](#), [22430](#)
 - \coffin_join:NnnNnnnn .. [228](#), [22817](#)
 - \coffin_log_structure:N .. [229](#), [23205](#)
 - \coffin_mark_handle:Nnnn .. [229](#), [23040](#)

- \coffin_new:N [226](#),
[933](#), [22460](#), [22577](#), [22579](#), [22580](#),
[22581](#), [22582](#), [22988](#), [22989](#), [22990](#)
- \coffin_resize:Nnn [237](#), [23862](#)
- \coffin_rotate:Nn [237](#), [23714](#)
- \coffin_scale:Nnn [237](#), [23891](#)
- \coffin_set_eq:NN
[226](#), [22568](#), [22855](#), [22877](#), [22906](#), [23116](#)
- \coffin_set_horizontal_pole:Nnn .
..... [227](#), [22620](#)
- \coffin_set_vertical_pole:Nnn ...
..... [227](#), [22620](#)
- \coffin_show_structure:N
..... [229](#), [229](#), [23205](#)
- \coffin_typeset:Nnnnn [228](#), [22980](#)
- \coffin_wd:N
..... [228](#), [22583](#), [23220](#), [23865](#), [23901](#)
- \c_empty_coffin ... [229](#), [22577](#), [22983](#)
- \l_tmpa_coffin [229](#), [22581](#)
- \l_tmpb_coffin [229](#), [22581](#)
- coffin internal commands:
- __coffin_align:NnnNnnnnN
 .. [22819](#), [22860](#), [22881](#), [22889](#), [22983](#)
- \l_coffin_aligned_coffin
 [22577](#), [22820](#),
 [22821](#), [22825](#), [22831](#), [22835](#), [22838](#),
 [22854](#), [22855](#), [22861](#), [22862](#), [22863](#),
 [22864](#), [22865](#), [22869](#), [22872](#), [22876](#),
 [22877](#), [22882](#), [22883](#), [22884](#), [22885](#),
 [22886](#), [22920](#), [22937](#), [22984](#), [22985](#),
 [23190](#), [23197](#), [23199](#), [23201](#), [23203](#)
- \l_coffin_aligned_internal_
 coffin [22577](#), [22899](#), [22906](#)
- \l_coffin_bottom_corner_dim ...
 [23710](#), [23734](#), [23738](#),
 [23814](#), [23825](#), [23826](#), [23846](#), [23854](#)
- \l_coffin_bounding_prop
 [23708](#), [23723](#), [23751](#),
 [23753](#), [23759](#), [23761](#), [23770](#), [23833](#)
- \l_coffin_bounding_shift_dim ...
 .. [23709](#), [23732](#), [23832](#), [23838](#), [23839](#)
- __coffin_calculate_intersection:Nnn
 [22705](#), [22891](#), [22894](#), [23183](#)
- __coffin_calculate_intersection:nnnnnnnn
 [22705](#), [23129](#)
- __coffin_calculate_intersection_
 aux:nnnnN [22705](#)
- \c_coffin_corners_prop
 [22399](#), [22467](#), [22602](#)
- \l_coffin_cos_fp
 [963](#), [965](#), [23706](#), [23717](#), [23797](#), [23806](#)
- __coffin_display_attach:Nnnnn [23095](#)
- \l_coffin_display_coffin
 [22988](#), [23116](#), [23122](#), [23192](#),
 [23193](#), [23198](#), [23200](#), [23202](#), [23203](#)
- \l_coffin_display_coord_coffin .
 [22988](#), [23054](#),
 [23074](#), [23090](#), [23137](#), [23157](#), [23176](#)
- \l_coffin_display_font_tl
 [23033](#), [23062](#), [23145](#)
- __coffin_display_handles_
 aux:nnnn [23095](#)
- __coffin_display_handles_
 aux:nnnnnn [23095](#)
- \l_coffin_display_handles_prop .
 .. [22991](#), [23065](#), [23069](#), [23148](#), [23152](#)
- \l_coffin_display_offset_dim ...
 .. [23028](#), [23091](#), [23092](#), [23177](#), [23178](#)
- \l_coffin_display_pole_coffin ..
 .. [22988](#), [23042](#), [23053](#), [23097](#), [23135](#)
- \l_coffin_display_poles_prop ...
 [23032](#), [23107](#),
 [23112](#), [23115](#), [23117](#), [23119](#), [23126](#)
- \l_coffin_display_x_dim
 [23030](#), [23132](#), [23187](#)
- \l_coffin_display_y_dim
 [23030](#), [23133](#), [23189](#)
- \l_coffin_error_bool
 [22420](#), [22709](#), [22713](#),
 [22727](#), [22742](#), [22773](#), [23128](#), [23130](#)
- __coffin_find_bounding_shift: ..
 [23726](#), [23830](#)
- __coffin_find_bounding_shift_
 aux:nn [23830](#)
- __coffin_find_corner_maxima:N ..
 [23725](#), [23810](#)
- __coffin_find_corner_maxima_
 aux:nn [23810](#)
- __coffin_get_pole:NnN
 [22589](#), [22707](#), [22708](#), [22947](#), [22948](#),
 [22951](#), [22952](#), [23109](#), [23110](#), [23113](#)
- __coffin_gset_eq_structure:NN [22606](#)
- __coffin_if_exist:NTF
 [22442](#), [22453](#), [22475](#), [22490](#), [22521](#),
 [22537](#), [22570](#), [22622](#), [22633](#), [23213](#)
- \l_coffin_internal_box
 [22396](#), [22504](#),
 [22510](#), [22515](#), [22551](#), [22557](#), [22562](#),
 [23728](#), [23737](#), [23739](#), [23740](#), [23742](#)
- \l_coffin_internal_dim
 [22396](#), [22826](#), [22828](#), [22829](#),
 [23758](#), [23760](#), [23764](#), [23896](#), [23899](#)
- \l_coffin_internal_tl ... [22396](#),
 [22918](#), [22919](#), [22921](#), [23066](#), [23067](#),
 [23070](#), [23071](#), [23079](#), [23084](#), [23149](#),
 [23150](#), [23153](#), [23154](#), [23163](#), [23168](#)

- \l__coffin_left_corner_dim [23710](#), [23732](#), [23741](#),
[23815](#), [23821](#), [23822](#), [23845](#), [23853](#)
- __coffin_mark_handle_aux:nnnnNnn [23040](#)
- __coffin_offset_corner:Nnnnn . [22927](#)
- __coffin_offset_corners:Nnn [22843](#), [22844](#), [22850](#), [22851](#), [22927](#)
- __coffin_offset_pole:Nnnnnnn . [22908](#)
- __coffin_offset_poles:Nnn [22841](#), [22842](#),
[22847](#), [22848](#), [22873](#), [22874](#), [22908](#)
- \l__coffin_offset_x_dim [22421](#), [22823](#), [22824](#), [22827](#),
[22839](#), [22841](#), [22843](#), [22849](#), [22852](#),
[22875](#), [22895](#), [22903](#), [23186](#), [23194](#)
- \l__coffin_offset_y_dim [22421](#), [22842](#), [22844](#), [22849](#), [22852](#),
[22875](#), [22897](#), [22904](#), [23188](#), [23195](#)
- \l__coffin_pole_a_tl [22423](#), [22707](#), [22712](#), [22947](#), [22950](#),
[22951](#), [22954](#), [23109](#), [23111](#), [23114](#)
- \l__coffin_pole_b_tl [22423](#),
[22708](#), [22712](#), [22948](#), [22950](#), [22952](#),
[22954](#), [23110](#), [23111](#), [23113](#), [23114](#)
- \c__coffin_poles_prop [22406](#), [22469](#), [22604](#)
- __coffin_reset_structure:N [22456](#), [22482](#), [22501](#),
[22527](#), [22548](#), [22599](#), [22831](#), [22865](#)
- __coffin_resize_common:Nnn [23872](#), [23875](#), [23902](#)
- \l__coffin_right_corner_dim [23710](#), [23741](#), [23813](#), [23823](#), [23824](#)
- __coffin_rotate_bounding:nnn [23724](#), [23767](#)
- __coffin_rotate_corner:Nnnn [23719](#), [23767](#)
- __coffin_rotate_pole:Nnnnnn [23721](#), [23779](#)
- __coffin_rotate_vector:nnNN [23769](#), [23775](#), [23781](#), [23782](#), [23791](#)
- __coffin_scale_corner:Nnnn [23878](#), [23913](#)
- __coffin_scale_pole:Nnnnnn [23880](#), [23913](#)
- __coffin_scale_vector:nnNN [23906](#), [23915](#), [23921](#)
- \l__coffin_scale_x_fp [23858](#), [23864](#),
[23881](#), [23893](#), [23895](#), [23901](#), [23909](#)
- \l__coffin_scale_y_fp ... [23858](#),
[23866](#), [23894](#), [23895](#), [23899](#), [23911](#)
- \l__coffin_scaled_total_height_-
dim [23860](#), [23898](#), [23903](#)
- \l__coffin_scaled_width_dim [23860](#), [23900](#), [23903](#)
- __coffin_set_bounding:N [23722](#), [23749](#)
- __coffin_set_eq_structure:NN [22573](#), [22606](#)
- __coffin_set_pole:Nnn [22505](#), [22552](#), [22620](#), [22920](#), [22960](#),
[22964](#), [22972](#), [22976](#), [23784](#), [23922](#)
- __coffin_shift_corner:Nnnn [23744](#), [23841](#)
- __coffin_shift_pole:Nnnnnn [23746](#), [23841](#)
- __coffin_show_structure:NN .. [23205](#)
- \l__coffin_sin_fp [963](#), [965](#), [23706](#), [23716](#), [23798](#), [23805](#)
- \l__coffin_slope_x_fp [22418](#), [22767](#), [22772](#), [22781](#), [22789](#)
- \l__coffin_slope_y_fp [22418](#), [22769](#), [22772](#), [22783](#), [22790](#)
- __coffin_to_value:N [22429](#), [22434](#), [22464](#), [22465](#), [22466](#),
[22468](#), [22592](#), [22601](#), [22603](#), [22608](#),
[22609](#), [22610](#), [22611](#), [22615](#), [22616](#),
[22617](#), [22618](#), [22644](#), [22652](#), [22655](#),
[22661](#), [22664](#), [22673](#), [22678](#), [22683](#),
[22690](#), [22697](#), [22835](#), [22869](#), [22871](#),
[22910](#), [22929](#), [22937](#), [23108](#), [23224](#),
[23718](#), [23720](#), [23743](#), [23745](#), [23776](#),
[23816](#), [23843](#), [23851](#), [23877](#), [23879](#),
[23884](#), [23887](#), [23916](#), [23930](#), [23937](#)
- \l__coffin_top_corner_dim [23710](#), [23738](#), [23812](#), [23827](#), [23828](#)
- __coffin_update_B:nnnnnnnnN . [22945](#)
- __coffin_update_corners:N [22484](#), [22503](#), [22529](#), [22550](#), [22650](#)
- __coffin_update_poles:N [22483](#), [22502](#),
[22528](#), [22549](#), [22671](#), [22838](#), [22872](#)
- __coffin_update_T:nnnnnnnnN . [22945](#)
- __coffin_update_vertical_-
poles:NNN [22854](#), [22876](#), [22945](#)
- \l__coffin_x_dim [22425](#), [22716](#), [22725](#), [22745](#), [22748](#),
[22755](#), [22764](#), [22775](#), [22795](#), [22892](#),
[22896](#), [22915](#), [22923](#), [23132](#), [23184](#),
[23769](#), [23771](#), [23775](#), [23777](#), [23781](#),
[23786](#), [23915](#), [23917](#), [23921](#), [23924](#)
- \l__coffin_x_prime_dim [22425](#), [22892](#),
[22896](#), [23184](#), [23187](#), [23783](#), [23787](#)
- __coffin_x_shift_corner:Nnnn [23885](#), [23928](#)
- __coffin_x_shift_pole:Nnnnnn [23888](#), [23928](#)

- \l__coffin_y_dim
 - [22425](#), [22717](#), [22730](#), [22733](#),
 - [22740](#), [22757](#), [22762](#), [22796](#), [22893](#),
 - [22898](#), [22916](#), [22923](#), [23133](#), [23185](#),
 - [23769](#), [23771](#), [23775](#), [23777](#), [23781](#),
 - [23786](#), [23915](#), [23917](#), [23921](#), [23924](#)
- \l__coffin_y_prime_dim
 - [22425](#), [22893](#),
 - [22898](#), [23185](#), [23189](#), [23783](#), [23788](#)
- \color [23048](#), [23060](#), [23103](#), [23143](#)
- color commands:
 - \color_ensure_current:
 - [230](#), [931](#), [22479](#), [22523](#), [23260](#)
 - \color_group_begin:
 - [230](#), [230](#), [21894](#),
 - [21899](#), [21905](#), [21913](#), [21919](#), [21928](#),
 - [21935](#), [21950](#), [21957](#), [21964](#), [21969](#),
 - [21980](#), [21982](#), [21986](#), [21991](#), [21997](#),
 - [22003](#), [22011](#), [22017](#), [22025](#), [22031](#),
 - [22040](#), [22047](#), [22062](#), [22069](#), [23254](#)
 - \color_group_end: [230](#), [230](#),
 - [21894](#), [21899](#), [21905](#), [21913](#), [21919](#),
 - [21941](#), [21964](#), [21969](#), [21980](#), [21982](#),
 - [21986](#), [21991](#), [21997](#), [22003](#), [22011](#),
 - [22017](#), [22025](#), [22031](#), [22053](#), [23254](#)
- color internal commands:
 - \l__color_current_tl
 - [950](#), [23254](#), [23263](#), [23265](#), [23280](#)
 - __color_select:n [23265](#), [23267](#)
 - __color_select:w [23267](#)
 - __color_select_cmyk:w [23267](#)
 - __color_select_gray:w [23267](#)
 - __color_select_rgb:w [23267](#)
 - __color_select_spot:w [23267](#)
- \columnwidth [22497](#), [22543](#)
- \copy [315](#)
- \copyfont [991](#)
- cos [192](#)
- cosd [193](#)
- cot [192](#)
- cotd [193](#)
- \count [164](#), [166](#), [167](#), [168](#),
 - [172](#), [173](#), [175](#), [176](#), [179](#), [181](#), [182](#),
 - [183](#), [184](#), [188](#), [189](#), [191](#), [192](#), [316](#), [7607](#)
- \countdef [317](#)
- \cr [318](#)
- \crampeddisplaystyle [891](#)
- \crampedscriptscriptstyle [893](#)
- \crampedscriptstyle [895](#)
- \crampedtextstyle [896](#)
- \crr [319](#)
- \cs [14092](#)
- cs commands:
 - \cs:w [16](#), [889](#), [889](#), [1416](#),
 - [1438](#), [1440](#), [1489](#), [2070](#), [2098](#), [2296](#),
 - [2360](#), [2470](#), [2510](#), [2519](#), [2521](#), [2525](#),
 - [2526](#), [2527](#), [2567](#), [2573](#), [2579](#), [2585](#),
 - [2605](#), [2607](#), [2612](#), [2619](#), [2620](#), [2704](#),
 - [2708](#), [2732](#), [2979](#), [5361](#), [5475](#), [6163](#),
 - [6233](#), [6442](#), [6444](#), [10001](#), [10335](#),
 - [10391](#), [10484](#), [10525](#), [11414](#), [11433](#),
 - [11444](#), [11458](#), [11741](#), [11760](#), [11827](#),
 - [12656](#), [12845](#), [12877](#), [13294](#), [13320](#),
 - [13333](#), [13369](#), [13937](#), [15622](#), [15890](#),
 - [16423](#), [20976](#), [20979](#), [21766](#), [24021](#)
 - \cs_end: [16](#), [342](#), [1416](#), [1438](#),
 - [1440](#), [1444](#), [1489](#), [2064](#), [2070](#), [2092](#),
 - [2098](#), [2223](#), [2296](#), [2360](#), [2470](#), [2510](#),
 - [2519](#), [2521](#), [2525](#), [2526](#), [2527](#), [2567](#),
 - [2573](#), [2579](#), [2585](#), [2605](#), [2607](#), [2612](#),
 - [2619](#), [2620](#), [2704](#), [2708](#), [2732](#), [2979](#),
 - [5361](#), [5475](#), [6163](#), [6170](#), [6213](#), [6223](#),
 - [6233](#), [6439](#), [6445](#), [6447](#), [6449](#), [6451](#),
 - [6453](#), [6455](#), [6457](#), [6459](#), [6461](#), [6463](#),
 - [6465](#), [10001](#), [10335](#), [10391](#), [10484](#),
 - [10525](#), [11087](#), [11414](#), [11434](#), [11445](#),
 - [11458](#), [11744](#), [11760](#), [11835](#), [12659](#),
 - [12849](#), [12881](#), [13300](#), [13326](#), [13339](#),
 - [13372](#), [13943](#), [15622](#), [15890](#), [16582](#),
 - [20842](#), [20994](#), [21766](#), [24019](#), [24021](#)
 - \cs_generate_from_arg_count:NNnn
 - [13](#), [2276](#), [2318](#)
 - \cs_generate_variant:Nn [9](#),
 - [22](#), [24](#), [25](#), [237](#), [237](#), [338](#), [2757](#),
 - [3021](#), [3023](#), [3025](#), [3027](#), [3061](#), [3074](#),
 - [3075](#), [3080](#), [3081](#), [3086](#), [3087](#), [3107](#),
 - [3108](#), [3125](#), [3126](#), [3158](#), [3159](#), [3160](#),
 - [3161](#), [3162](#), [3163](#), [3188](#), [3189](#), [3190](#),
 - [3191](#), [3192](#), [3193](#), [3194](#), [3195](#), [3220](#),
 - [3221](#), [3222](#), [3223](#), [3224](#), [3225](#), [3226](#),
 - [3227](#), [3270](#), [3271](#), [3272](#), [3273](#), [3341](#),
 - [3342](#), [3343](#), [3344](#), [3406](#), [3407](#), [3412](#),
 - [3413](#), [3547](#), [3565](#), [3579](#), [3595](#), [3600](#),
 - [3602](#), [3611](#), [3623](#), [3624](#), [3643](#), [3646](#),
 - [3651](#), [3652](#), [3752](#), [3763](#), [3764](#), [3786](#),
 - [3796](#), [3931](#), [3933](#), [3935](#), [3971](#), [3986](#),
 - [3987](#), [3990](#), [3991](#), [4002](#), [4024](#), [4025](#),
 - [4026](#), [4027](#), [4060](#), [4061](#), [4066](#), [4067](#),
 - [4156](#), [4214](#), [4232](#), [4258](#), [4309](#), [4370](#),
 - [4448](#), [4467](#), [4505](#), [4520](#), [4537](#), [4538](#),
 - [4539](#), [4552](#), [4594](#), [4640](#), [4641](#), [4730](#),
 - [4733](#), [4736](#), [4739](#), [4742](#), [4771](#), [4772](#),
 - [4773](#), [4774](#), [4775](#), [4776](#), [4812](#), [4813](#),
 - [4818](#), [4819](#), [4841](#), [4842](#), [4843](#), [4844](#),
 - [4849](#), [4850](#), [4851](#), [4852](#), [4869](#), [4870](#),
 - [4895](#), [4896](#), [4913](#), [4914](#), [4974](#), [4987](#),

- 4988, 5006, 5032, 5033, 5085, 5105,
 5134, 5145, 5146, 5170, 5193, 5207,
 5241, 5243, 5364, 5389, 5407, 5408,
 5413, 5414, 5417, 5420, 5445, 5446,
 5447, 5448, 5461, 5462, 5463, 5464,
 5471, 5472, 5474, 6106, 6110, 6242,
 6255, 6256, 6257, 6258, 6261, 6262,
 6269, 6270, 6287, 6289, 6399, 6400,
 6405, 6406, 6627, 6667, 6668, 6669,
 6670, 6684, 6685, 6694, 6695, 6705,
 6706, 6707, 6708, 6718, 6719, 6720,
 6721, 6732, 6757, 6758, 6816, 6817,
 6855, 6856, 6861, 6862, 6945, 6979,
 7003, 7016, 7056, 7065, 7089, 7096,
 7126, 7128, 7269, 7270, 7271, 7272,
 7884, 7887, 7890, 7893, 7896, 7911,
 7914, 7917, 7984, 7985, 7986, 7987,
 7994, 7995, 8014, 8015, 8016, 8017,
 8030, 8066, 8068, 8070, 8072, 8089,
 8090, 8152, 8167, 8173, 8175, 8657,
 9077, 9078, 9079, 9080, 9108, 9113,
 9146, 9164, 9275, 9300, 9307, 9319,
 9334, 9337, 9355, 9769, 9812, 10004,
 10011, 10017, 10018, 10023, 10024,
 10043, 10044, 10048, 10052, 10062,
 10063, 10073, 10074, 10334, 10373,
 10394, 10401, 10406, 10407, 10412,
 10413, 10432, 10433, 10435, 10437,
 10444, 10445, 10452, 10453, 10483,
 10505, 10506, 10508, 10528, 10535,
 10542, 10543, 10548, 10549, 10574,
 10575, 10578, 10581, 10588, 10589,
 10596, 10597, 10608, 10610, 10805,
 10904, 10920, 10984, 11098, 11231,
 11232, 11235, 11243, 11251, 11260,
 11268, 11276, 11290, 11308, 11341,
 11481, 12109, 12112, 13766, 13834,
 13841, 13842, 13843, 13846, 13847,
 13850, 13851, 13856, 13857, 13864,
 13865, 13866, 13867, 13869, 13871,
 14087, 14149, 17063, 17117, 17195,
 17240, 17255, 17310, 17757, 17759,
 17781, 17784, 17790, 17796, 18485,
 19203, 21769, 21774, 21775, 21780,
 21781, 21788, 21789, 21796, 21797,
 21805, 21806, 21807, 21814, 21815,
 21816, 21819, 21820, 21850, 21851,
 21859, 21862, 21865, 21874, 21892,
 21907, 21908, 21921, 21922, 21937,
 21938, 21959, 21960, 21977, 21978,
 22005, 22006, 22019, 22020, 22033,
 22034, 22049, 22050, 22071, 22072,
 22075, 22076, 22219, 22256, 22271,
 22283, 22299, 22309, 22326, 22329,
 22459, 22472, 22487, 22518, 22534,
 22567, 22576, 22647, 22648, 22649,
 22857, 22888, 22987, 23094, 23180,
 23207, 23210, 23269, 23597, 23643,
 23690, 23704, 23748, 23874, 23905,
 24042, 24043, 24044, 24045, 24081,
 24092, 24106, 24116, 24146, 24186,
 24192, 24296, 24309, 24365, 24366,
 24392, 24393, 24478, 24513, 25443,
 25447, 25497, 25499, 25502, 25506,
 25604, 25700, 25755, 25841, 25853,
 25926, 26128, 26134, 26176, 26196,
 26330, 26429, 26486, 26502, 26526,
 26660, 26769, 26776, 26878, 26909
 \cs_gset:Nn 13, 2291, 2355
 \cs_gset:Npn 9,
 11, 291, 1473, 2171, 2185, 2187,
 5109, 7336, 8229, 8231, 8269, 12535
 \cs_gset:Npx 11,
 1473, 2172, 2185, 2188, 5114, 20850
 \cs_gset_eq:NN 14,
 2203, 2220, 2228, 3059, 3100, 3105,
 4728, 5119, 5124, 6150, 6251, 6254,
 7267, 7882, 8155, 8163, 9161, 9316
 \cs_gset_nopar:Nn 13, 2291, 2355
 \cs_gset_nopar:Npn
 11, 1473, 1667, 2169, 2177, 2181, 4602
 \cs_gset_nopar:Npx
 11, 1004, 1473, 2170, 2177,
 2182, 3066, 3072, 3151, 3154, 3157,
 3178, 3181, 3184, 3187, 3210, 3213,
 3216, 3219, 25382, 25411, 25417, 25446
 \cs_gset_protected:Nn 13, 2291, 2355
 \cs_gset_protected:Npn
 11, 1473, 1775, 2175,
 2197, 2199, 3569, 4218, 5724, 6965,
 8158, 9221, 10308, 14154, 17697,
 17705, 17718, 18115, 18383, 18388
 \cs_gset_protected:Npx .. 11, 1473,
 2176, 2197, 2200, 5735, 10315, 14161
 \cs_gset_protected_nopar:Nn
 13, 2291, 2355
 \cs_gset_protected_nopar:Npn ...
 11, 1473, 2173, 2191, 2193
 \cs_gset_protected_nopar:Npx ...
 11, 1473, 2174, 2191, 2194
 \cs_if_eq:NNTF 20,
 2387, 2394, 2395, 2398, 2399, 2402,
 2403, 8462, 11053, 12308, 12318,
 12344, 12346, 12348, 12540, 19827
 \cs_if_eq_p:NN 20, 2387, 19843
 \cs_if_exist 213
 \cs_if_exist:N 20,
 3127, 3128, 4820, 4822, 5421, 5423,

- 6299, 6301, 6686, 6688, 8091, 8093,
10025, 10027, 10414, 10416, 10550,
10552, 13896, 13897, 21798, 21800
- \cs_if_exist:NTF
..... 15, 20, 288, 477, 506, 1596,
1605, 2050, 2107, 2109, 2111, 2113,
2115, 2117, 2119, 2121, 2407, 3046,
4077, 5391, 5392, 5394, 5395, 6139,
6140, 6207, 6545, 6546, 6547, 6555,
7572, 7591, 8195, 9096, 9100, 9180,
9263, 9267, 9665, 9688, 9746, 10819,
10927, 10978, 11357, 11399, 11411,
11424, 11430, 11441, 11456, 11494,
11502, 12533, 12708, 13823, 17350,
17366, 17695, 17713, 17716, 19562,
20965, 22432, 22434, 23583, 24193,
24261, 24686, 24706, 24713, 25835,
25844, 26171, 26179, 26185, 26191
- \cs_if_exist_p:N
..... 20, 288, 288, 2050, 6550, 24259
- \cs_if_exist_use:N
..... 15, 315, 2106, 11413, 20009
- \cs_if_exist_use:NTF
..... 15, 1541, 1549, 2106,
2108, 2110, 2116, 2118, 3010, 6565,
11019, 11990, 12666, 12668, 18807,
18814, 19166, 19171, 19208, 19630,
19715, 20899, 24624, 24637, 24639
- \cs_if_free:NTF 20, 92,
506, 2078, 2152, 2932, 2946, 2975, 8527
- \cs_if_free_p:N 19, 20, 92, 2078
- \cs_log:N 15, 324, 2432
- \cs_meaning:N
..... 14, 295, 1425, 1441, 1449, 2444
- \cs_new:Nn 11, 93, 2291, 2355
- \cs_new:Npn
.. 9, 10, 13, 92, 92, 291, 292, 303,
305, 795, 1747, 1791, 2161, 2185,
2189, 2262, 2264, 2266, 2274, 2326,
2392, 2393, 2394, 2395, 2396, 2397,
2398, 2399, 2400, 2401, 2402, 2403,
2446, 2449, 2458, 2459, 2463, 2464,
2465, 2466, 2467, 2468, 2469, 2471,
2473, 2486, 2492, 2498, 2509, 2511,
2518, 2520, 2522, 2529, 2530, 2532,
2534, 2536, 2541, 2546, 2552, 2558,
2564, 2570, 2576, 2582, 2588, 2595,
2602, 2609, 2616, 2624, 2625, 2626,
2627, 2628, 2629, 2630, 2631, 2632,
2633, 2634, 2641, 2642, 2643, 2644,
2645, 2646, 2647, 2648, 2649, 2650,
2651, 2652, 2653, 2654, 2655, 2656,
2665, 2666, 2668, 2673, 2678, 2688,
2689, 2691, 2693, 2695, 2697, 2703,
2705, 2711, 2713, 2720, 2721, 2722,
2723, 2724, 2727, 2729, 2732, 2733,
2734, 2736, 2741, 2751, 2754, 2825,
2846, 2866, 2869, 2878, 2891, 2906,
2919, 2977, 3052, 3268, 3439, 3501,
3504, 3505, 3506, 3507, 3518, 3519,
3524, 3529, 3534, 3539, 3541, 3550,
3552, 3558, 3560, 3596, 3598, 3601,
3603, 3612, 3617, 3622, 3625, 3632,
3639, 3641, 3644, 3655, 3667, 3675,
3681, 3687, 3693, 3700, 3711, 3720,
3722, 3729, 3735, 3737, 3739, 3753,
3755, 3757, 3765, 3770, 3775, 3787,
3788, 3789, 3797, 3841, 3850, 3881,
3902, 3909, 3917, 3923, 3930, 3955,
4076, 4092, 4134, 4139, 4144, 4149,
4154, 4159, 4165, 4170, 4175, 4180,
4185, 4187, 4193, 4195, 4203, 4205,
4207, 4233, 4259, 4261, 4263, 4274,
4283, 4286, 4297, 4306, 4308, 4310,
4318, 4320, 4327, 4348, 4358, 4363,
4368, 4369, 4371, 4379, 4381, 4389,
4395, 4401, 4420, 4422, 4431, 4437,
4444, 4446, 4449, 4459, 4466, 4468,
4476, 4481, 4486, 4497, 4504, 4506,
4512, 4514, 4519, 4521, 4527, 4528,
4533, 4534, 4535, 4536, 4540, 4545,
4550, 4553, 4555, 4563, 4568, 4610,
4616, 4624, 4631, 4638, 4642, 4648,
4677, 4683, 4686, 4711, 4716, 4804,
4810, 4840, 4853, 4908, 4942, 4972,
5000, 5005, 5027, 5062, 5064, 5072,
5078, 5086, 5088, 5090, 5099, 5147,
5165, 5169, 5171, 5194, 5195, 5196,
5203, 5205, 5268, 5270, 5273, 5279,
5307, 5320, 5328, 5341, 5348, 5355,
5475, 5482, 5496, 5501, 5507, 5518,
5523, 5530, 5532, 5534, 5536, 5538,
5540, 5542, 5558, 5563, 5568, 5573,
5578, 5580, 5586, 5608, 5616, 5624,
5630, 5636, 5644, 5652, 5658, 5679,
5686, 5702, 5712, 5714, 5750, 5764,
5770, 5802, 5834, 5836, 5838, 5844,
5850, 5862, 5870, 5882, 5890, 5923,
5956, 5958, 5960, 5962, 5964, 5969,
5974, 5979, 5984, 5985, 5986, 5987,
5988, 5989, 5990, 5991, 5992, 5993,
5994, 5995, 5996, 5997, 5998, 5999,
6000, 6009, 6010, 6019, 6025, 6027,
6036, 6043, 6049, 6051, 6053, 6069,
6080, 6103, 6162, 6194, 6220, 6221,
6230, 6231, 6284, 6311, 6319, 6329,
6334, 6339, 6341, 6349, 6350, 6351,
6352, 6353, 6354, 6355, 6356, 6357,

6360, 6375, 6388, 6389, 6395, 6397,
6401, 6403, 6407, 6415, 6420, 6428,
6434, 6441, 6443, 6445, 6446, 6448,
6450, 6452, 6454, 6456, 6458, 6460,
6462, 6464, 6466, 6471, 6472, 6473,
6474, 6475, 6476, 6477, 6478, 6479,
6480, 6490, 6492, 6497, 6499, 6501,
6504, 6506, 6579, 6585, 6591, 6620,
6621, 6660, 6756, 6852, 6854, 6863,
6870, 6873, 6886, 6892, 6930, 6939,
6946, 6952, 6959, 7004, 7006, 7008,
7026, 7033, 7057, 7058, 7061, 7063,
7066, 7074, 7090, 7097, 7105, 7107,
7121, 7123, 7124, 7155, 7225, 7231,
7237, 7243, 7274, 7280, 7321, 7329,
7401, 7521, 7551, 7626, 7638, 7639,
7647, 7656, 7665, 7678, 7679, 7680,
7681, 7721, 7729, 7731, 7733, 7743,
7753, 7839, 7842, 7851, 7860, 7875,
7918, 7924, 7931, 7936, 7938, 7943,
7945, 7971, 8018, 8024, 8109, 8115,
8137, 8146, 8168, 8170, 8268, 8354,
8359, 8364, 8369, 8374, 8379, 8466,
8506, 8508, 8525, 8658, 9034, 9043,
9048, 9057, 9062, 9067, 9072, 9205,
9207, 9360, 9368, 9412, 9458, 9509,
9518, 9537, 9545, 9551, 9559, 9569,
9574, 9580, 9586, 9648, 9651, 9737,
9901, 10077, 10082, 10089, 10102,
10110, 10118, 10120, 10138, 10144,
10157, 10159, 10161, 10163, 10165,
10173, 10178, 10183, 10188, 10193,
10195, 10201, 10203, 10211, 10219,
10225, 10231, 10239, 10247, 10253,
10274, 10281, 10295, 10331, 10335,
10338, 10345, 10353, 10362, 10364,
10475, 10480, 10484, 10493, 10503,
10605, 10757, 10765, 10767, 10887,
11454, 11472, 11477, 11479, 11482,
11490, 11576, 11577, 11578, 11579,
11580, 11581, 11582, 11583, 11584,
11585, 11616, 11618, 11620, 11629,
11631, 11643, 11644, 11646, 11656,
11666, 11676, 11686, 11694, 11696,
11703, 11705, 11706, 11711, 11718,
11732, 11734, 11750, 11751, 11759,
11761, 11770, 11772, 11784, 11789,
11793, 11798, 11800, 11802, 11804,
11806, 11813, 11815, 11823, 11825,
11837, 11839, 11841, 11843, 11867,
11869, 11871, 11872, 11873, 11875,
11877, 11879, 11881, 11899, 11914,
11915, 11921, 11936, 11943, 11949,
11964, 11970, 12096, 12097, 12098,
12099, 12100, 12101, 12102, 12107,
12110, 12156, 12158, 12160, 12162,
12168, 12172, 12174, 12183, 12184,
12193, 12206, 12219, 12226, 12240,
12256, 12268, 12279, 12289, 12295,
12306, 12316, 12342, 12353, 12362,
12373, 12378, 12398, 12400, 12411,
12416, 12429, 12451, 12452, 12456,
12473, 12474, 12498, 12506, 12524,
12555, 12581, 12585, 12588, 12590,
12596, 12608, 12620, 12627, 12633,
12641, 12664, 12679, 12698, 12706,
12721, 12736, 12747, 12757, 12767,
12772, 12781, 12798, 12811, 12816,
12822, 12824, 12831, 12861, 12889,
12905, 12916, 12921, 12939, 12957,
12968, 12983, 12988, 12998, 13008,
13018, 13034, 13082, 13087, 13094,
13102, 13108, 13113, 13117, 13134,
13142, 13174, 13191, 13205, 13224,
13232, 13241, 13250, 13261, 13263,
13277, 13287, 13288, 13305, 13312,
13317, 13330, 13343, 13348, 13395,
13396, 13400, 13417, 13439, 13441,
13452, 13482, 13486, 13501, 13518,
13542, 13544, 13546, 13548, 13558,
13563, 13574, 13586, 13597, 13610,
13630, 13648, 13650, 13662, 13668,
13676, 13690, 13697, 13708, 13715,
13729, 13737, 13753, 13764, 13767,
13777, 13788, 13791, 13892, 13894,
13903, 13925, 13930, 13947, 13974,
13975, 13976, 13977, 13993, 14004,
14012, 14024, 14030, 14036, 14044,
14052, 14058, 14064, 14072, 14080,
14098, 14111, 14133, 14181, 14187,
14198, 14222, 14224, 14226, 14228,
14236, 14240, 14247, 14254, 14255,
14256, 14257, 14258, 14259, 14262,
14264, 14293, 14301, 14312, 14314,
14316, 14323, 14347, 14349, 14359,
14374, 14383, 14397, 14405, 14413,
14420, 14427, 14435, 14445, 14459,
14470, 14471, 14477, 14494, 14501,
14503, 14510, 14515, 14532, 14533,
14534, 14553, 14559, 14569, 14581,
14588, 14602, 14610, 14648, 14657,
14678, 14680, 14682, 14691, 14702,
14714, 14729, 14742, 14755, 14763,
14781, 14799, 14806, 14814, 14824,
14825, 14834, 14835, 14844, 14854,
14868, 14878, 14889, 14897, 14899,
14910, 14916, 14951, 14972, 14974,
14976, 14978, 14985, 14994, 14999,

15006, 15013, 15033, 15038, 15055,
 15065, 15067, 15077, 15084, 15086,
 15092, 15094, 15096, 15100, 15119,
 15120, 15125, 15133, 15134, 15157,
 15170, 15177, 15185, 15186, 15187,
 15188, 15189, 15190, 15198, 15204,
 15206, 15208, 15230, 15235, 15245,
 15255, 15266, 15279, 15290, 15295,
 15302, 15311, 15313, 15322, 15331,
 15345, 15347, 15349, 15362, 15372,
 15377, 15386, 15394, 15401, 15407,
 15416, 15418, 15430, 15435, 15443,
 15448, 15458, 15464, 15470, 15477,
 15484, 15486, 15491, 15493, 15498,
 15500, 15514, 15524, 15536, 15541,
 15548, 15558, 15560, 15571, 15585,
 15599, 15619, 15632, 15634, 15639,
 15652, 15657, 15665, 15670, 15680,
 15692, 15722, 15723, 15724, 15726,
 15728, 15730, 15744, 15750, 15759,
 15778, 15784, 15794, 15813, 15821,
 15854, 15860, 15869, 15871, 15885,
 15887, 15895, 15897, 15913, 15929,
 15945, 15961, 15977, 15993, 15998,
 16023, 16046, 16075, 16091, 16101,
 16112, 16133, 16148, 16153, 16158,
 16160, 16174, 16176, 16191, 16199,
 16209, 16219, 16253, 16255, 16257,
 16259, 16261, 16276, 16291, 16306,
 16321, 16336, 16351, 16359, 16373,
 16375, 16381, 16393, 16401, 16408,
 16584, 16592, 16603, 16604, 16606,
 16607, 16618, 16625, 16627, 16633,
 16644, 16654, 16661, 16668, 16683,
 16722, 16735, 16766, 16772, 16779,
 16799, 16801, 16818, 16833, 16846,
 16853, 16858, 16860, 16869, 16882,
 16885, 16906, 16919, 16934, 16952,
 16967, 16977, 16986, 16999, 17015,
 17032, 17045, 17051, 17053, 17058,
 17059, 17060, 17061, 17064, 17069,
 17075, 17080, 17082, 17105, 17113,
 17115, 17118, 17123, 17129, 17134,
 17136, 17159, 17184, 17193, 17194,
 17196, 17201, 17203, 17208, 17210,
 17220, 17228, 17236, 17238, 17241,
 17246, 17251, 17253, 17254, 17256,
 17261, 17266, 17268, 17274, 17281,
 17295, 17300, 17302, 17312, 17314,
 17316, 17318, 17346, 17348, 17354,
 17360, 17368, 17373, 17379, 17385,
 17402, 17407, 17417, 17427, 17429,
 17435, 17436, 17449, 17464, 17476,
 17481, 17490, 17492, 17499, 17509,
 17515, 17538, 17543, 17552, 17576,
 17577, 17591, 17624, 17626, 17632,
 17771, 17914, 17927, 17932, 17938,
 17939, 17946, 17953, 17960, 17967,
 17974, 17975, 17977, 17984, 17990,
 18073, 18078, 18079, 18087, 18093,
 18270, 18275, 18282, 18319, 18324,
 18339, 18362, 18367, 18415, 18421,
 18439, 18444, 18459, 18461, 18463,
 18470, 18515, 18544, 18549, 18805,
 18811, 18822, 18827, 18840, 18845,
 18857, 18869, 18879, 18888, 18908,
 19007, 19025, 19033, 19045, 19057,
 19551, 19825, 19839, 19880, 19921,
 20081, 20232, 20617, 20749, 20751,
 20757, 20758, 20765, 20775, 20931,
 21301, 21306, 21704, 21760, 23291,
 23292, 23296, 23297, 23692, 23694,
 23699, 24017, 24025, 24034, 24036,
 24038, 24040, 24046, 24083, 24091,
 24093, 24100, 24107, 24117, 24125,
 24127, 24133, 24139, 24181, 24244,
 24260, 24321, 24334, 24340, 24345,
 24357, 24358, 24359, 24411, 24413,
 24415, 24421, 24423, 24429, 24439,
 24444, 24473, 24475, 24479, 24480,
 24481, 24482, 24483, 24484, 24485,
 24493, 24500, 24511, 24514, 24520,
 24524, 24538, 24550, 24555, 24563,
 24577, 24582, 24593, 24604, 24610,
 24615, 24622, 24634, 24649, 24656,
 24678, 24680, 24682, 24684, 24695,
 24704, 24730, 24741, 24750, 24764,
 24770, 24773, 24788, 24794, 24796,
 24802, 24816, 24820, 24826, 24836,
 24842, 24860, 24870, 24877, 24897,
 24907, 24920, 24928, 24934, 24954,
 24963, 24969, 24991, 25004, 25013,
 25019, 25035, 25319, 25329, 25335,
 25463, 25471, 25491, 25496, 25498,
 25500, 25501, 25503, 25504, 25507,
 25509, 25517, 25529, 25538, 25547,
 25549, 25551, 25559, 25567, 25572,
 25578, 25601, 25602, 25603, 25605,
 26073, 26515, 26601, 26875, 27034
 \cs_new:Npx 10, 30, 30,
 336, 337, 2161, 2185, 2190, 2784,
 2962, 3048, 7017, 7027, 9406, 11725,
 12567, 13154, 13378, 14318, 16240,
 16246, 16420, 16778, 17330, 18295,
 18828, 18830, 18832, 18834, 18836,
 18838, 24057, 24059, 24061, 24068
 \cs_new_eq:NN 14, 93, 316, 318, 472,
 1962, 2203, 2448, 2457, 2479, 2731,

- 3104, 3105, 3400, 3401, 3970, 3984,
 3985, 3988, 3989, 4055, 4592, 4593,
 4707, 4723, 4743, 4744, 4745, 4746,
 4747, 4748, 4749, 4750, 5208, 5209,
 5210, 5211, 5212, 5213, 5214, 5215,
 5216, 5217, 5218, 5219, 5220, 5221,
 5222, 5223, 5224, 5225, 5226, 5227,
 5228, 5229, 5230, 5231, 5232, 5233,
 5261, 5262, 5263, 5264, 5265, 5396,
 5397, 5400, 5473, 5749, 6105, 6109,
 6152, 6156, 6238, 6239, 6241, 6259,
 6260, 6531, 6532, 6533, 6534, 6537,
 6538, 6539, 6540, 6576, 6623, 6624,
 6628, 6629, 6630, 6631, 6632, 6633,
 6634, 6635, 6636, 6637, 6638, 6639,
 6640, 6641, 6642, 6643, 6784, 6785,
 6786, 6787, 6788, 6789, 6790, 6791,
 6792, 6793, 6794, 6795, 6796, 6797,
 6798, 6799, 7273, 7335, 7422, 7424,
 7425, 7426, 7428, 7431, 7432, 7674,
 7675, 7676, 7869, 7870, 7897, 7898,
 7899, 7900, 7901, 7902, 7903, 7904,
 9107, 9177, 9274, 9361, 9828, 9836,
 9994, 9995, 9996, 10333, 10372,
 10376, 10377, 10482, 10485, 10495,
 10507, 10511, 10512, 10607, 10609,
 10613, 10614, 11573, 11574, 11575,
 11769, 12239, 12267, 12275, 12276,
 12277, 12286, 12288, 13081, 13200,
 13201, 13202, 13833, 13844, 13845,
 17309, 17311, 17359, 18066, 18067,
 18506, 18518, 18640, 18641, 18741,
 18749, 18770, 18818, 18819, 18820,
 18821, 18998, 20509, 21109, 21752,
 21802, 21803, 21804, 21817, 21818,
 21829, 21830, 21831, 21944, 21975,
 21976, 22056, 22073, 22074, 22429,
 22583, 22584, 22585, 22586, 22587,
 22588, 23254, 23286, 23287, 23288,
 24241, 24536, 24632, 24855, 24926,
 24927, 25394, 25395, 25697, 25922,
 25925, 26267, 26324, 26325, 26377,
 26385, 26400, 26471, 26525, 26531,
 26532, 26655, 26658, 26862, 26863,
 26874, 26877, 27075, 27086, 27112
 \cs_new_nopar:Nn 12, 2291, 2355
 \cs_new_nopar:Npn 10,
 316, 317, 2161, 2177, 2183, 3691, 3692
 \cs_new_nopar:Npx 10, 2161, 2177, 2184
 \cs_new_protected:Nn . 12, 2291, 2355
 \cs_new_protected:Npn
 10, 303, 1744, 2161,
 2197, 2201, 2203, 2204, 2205, 2206,
 2207, 2208, 2209, 2210, 2211, 2216,
 2217, 2218, 2219, 2221, 2276, 2286,
 2288, 2299, 2308, 2405, 2414, 2416,
 2418, 2420, 2422, 2430, 2432, 2433,
 2435, 2436, 2438, 2480, 2623, 2635,
 2636, 2637, 2638, 2639, 2640, 2657,
 2658, 2659, 2660, 2661, 2662, 2663,
 2664, 2683, 2726, 2748, 2758, 2771,
 2789, 2793, 2796, 2805, 2917, 2942,
 2971, 2982, 2990, 3001, 3020, 3022,
 3024, 3026, 3028, 3044, 3056, 3063,
 3069, 3076, 3078, 3082, 3084, 3090,
 3096, 3115, 3123, 3141, 3144, 3147,
 3150, 3153, 3156, 3165, 3168, 3171,
 3174, 3177, 3180, 3183, 3186, 3197,
 3200, 3203, 3206, 3209, 3212, 3215,
 3218, 3229, 3231, 3233, 3235, 3258,
 3276, 3288, 3290, 3304, 3333, 3335,
 3337, 3339, 3345, 3367, 3373, 3402,
 3404, 3408, 3410, 3475, 3476, 3477,
 3566, 3577, 3580, 3586, 3588, 3647,
 3649, 3759, 3761, 3932, 3934, 3936,
 3941, 3943, 3956, 4016, 4018, 4020,
 4022, 4028, 4043, 4056, 4058, 4062,
 4064, 4215, 4230, 4239, 4249, 4251,
 4599, 4698, 4725, 4731, 4734, 4737,
 4740, 4751, 4756, 4761, 4766, 4777,
 4779, 4781, 4814, 4816, 4824, 4832,
 4845, 4847, 4855, 4857, 4859, 4871,
 4873, 4875, 4897, 4899, 4901, 4946,
 4954, 4964, 4975, 4977, 4979, 4981,
 4989, 5007, 5009, 5011, 5106, 5111,
 5116, 5122, 5128, 5135, 5240, 5242,
 5244, 5358, 5369, 5404, 5406, 5409,
 5411, 5416, 5419, 5434, 5437, 5440,
 5443, 5450, 5453, 5456, 5459, 5466,
 5469, 5476, 5716, 5718, 5720, 5727,
 5729, 5731, 5743, 6107, 6111, 6160,
 6167, 6168, 6179, 6181, 6182, 6183,
 6241, 6244, 6247, 6250, 6253, 6264,
 6267, 6280, 6282, 6286, 6288, 6290,
 6527, 6578, 6625, 6644, 6646, 6648,
 6671, 6673, 6675, 6690, 6692, 6696,
 6698, 6700, 6709, 6711, 6713, 6722,
 6730, 6733, 6735, 6737, 6745, 6773,
 6802, 6804, 6806, 6818, 6820, 6822,
 6857, 6859, 6903, 6960, 6974, 6980,
 6991, 6996, 7125, 7127, 7129, 7139,
 7140, 7141, 7153, 7157, 7159, 7161,
 7163, 7165, 7167, 7169, 7171, 7173,
 7175, 7177, 7179, 7181, 7183, 7185,
 7187, 7189, 7191, 7193, 7195, 7197,
 7199, 7201, 7203, 7205, 7207, 7209,
 7211, 7213, 7215, 7217, 7219, 7221,
 7223, 7227, 7229, 7233, 7235, 7239,

7241, 7245, 7257, 7682, 7684, 7686,
7691, 7709, 7711, 7713, 7715, 7717,
7719, 7762, 7870, 7879, 7885, 7888,
7891, 7894, 7909, 7912, 7915, 7961,
7963, 7972, 7978, 7988, 7996, 8005,
8053, 8054, 8055, 8074, 8076, 8078,
8153, 8172, 8174, 8176, 8203, 8211,
8216, 8218, 8225, 8227, 8234, 8275,
8287, 8292, 8309, 8339, 8345, 8389,
8391, 8460, 8496, 8510, 8535, 8557,
8558, 8571, 8576, 8602, 8611, 8613,
8615, 8632, 8659, 8661, 8663, 8665,
8672, 9107, 9111, 9127, 9130, 9147,
9153, 9165, 9166, 9167, 9194, 9196,
9209, 9211, 9213, 9219, 9226, 9274,
9279, 9283, 9301, 9308, 9320, 9321,
9322, 9332, 9335, 9338, 9344, 9350,
9357, 9359, 9369, 9400, 9418, 9444,
9467, 9479, 9498, 9502, 9590, 9606,
9614, 9623, 9630, 9636, 9710, 9724,
9739, 9770, 9785, 9791, 9798, 9813,
9830, 9838, 9844, 9849, 9862, 9883,
9884, 9885, 9961, 9969, 9975, 9982,
9984, 9986, 9988, 9990, 9998, 10006,
10013, 10015, 10019, 10021, 10038,
10041, 10046, 10050, 10054, 10057,
10065, 10068, 10167, 10304, 10311,
10323, 10374, 10378, 10388, 10396,
10403, 10405, 10408, 10410, 10427,
10430, 10434, 10436, 10439, 10442,
10447, 10450, 10509, 10513, 10522,
10530, 10537, 10540, 10544, 10546,
10569, 10572, 10577, 10580, 10583,
10586, 10591, 10594, 10611, 10615,
10629, 10646, 10652, 10657, 10670,
10676, 10681, 10695, 10698, 10708,
10720, 10747, 10758, 10760, 10797,
10799, 10806, 10811, 10816, 10829,
10835, 10859, 10871, 10889, 10905,
10921, 10923, 10925, 10944, 10955,
10957, 10959, 10982, 10985, 10999,
11012, 11017, 11021, 11029, 11031,
11041, 11069, 11078, 11087, 11088,
11099, 11101, 11103, 11105, 11107,
11109, 11111, 11113, 11115, 11117,
11119, 11121, 11123, 11125, 11127,
11129, 11131, 11133, 11135, 11137,
11139, 11141, 11143, 11145, 11147,
11149, 11151, 11153, 11155, 11157,
11159, 11161, 11163, 11165, 11167,
11169, 11171, 11173, 11175, 11177,
11179, 11181, 11183, 11185, 11187,
11189, 11191, 11193, 11195, 11197,
11199, 11201, 11203, 11205, 11207,
11209, 11211, 11213, 11215, 11217,
11219, 11221, 11223, 11225, 11233,
11236, 11244, 11252, 11258, 11261,
11269, 11277, 11283, 11291, 11297,
11299, 11309, 11315, 11320, 11325,
11342, 11355, 11369, 11395, 11409,
11419, 11463, 11507, 11509, 11511,
11586, 11589, 11690, 11988, 12005,
12007, 12009, 12011, 12039, 12041,
12043, 12045, 12065, 12067, 12069,
12071, 12073, 12075, 12077, 12079,
12081, 13744, 13751, 13832, 13835,
13837, 13839, 13848, 13849, 13852,
13854, 13858, 13859, 13860, 13861,
13862, 13868, 13870, 13872, 13877,
13879, 14150, 14157, 14169, 16811,
17565, 17596, 17598, 17605, 17616,
17654, 17673, 17681, 17693, 17730,
17737, 17756, 17758, 17760, 17779,
17782, 17785, 17791, 17797, 17812,
17821, 17841, 17851, 17861, 17867,
17873, 17874, 17881, 17884, 17894,
17904, 18000, 18007, 18009, 18025,
18059, 18061, 18095, 18106, 18124,
18134, 18136, 18160, 18167, 18179,
18182, 18185, 18195, 18203, 18210,
18219, 18234, 18251, 18262, 18372,
18379, 18381, 18399, 18409, 18497,
18501, 18513, 18516, 18519, 18521,
18530, 18536, 18542, 18573, 18575,
18576, 18581, 18587, 18595, 18607,
18623, 18642, 18652, 18660, 18662,
18670, 18682, 18702, 18704, 18709,
18717, 18719, 18726, 18731, 18733,
18735, 18742, 18750, 18752, 18754,
18756, 18763, 18768, 18771, 18786,
19001, 19065, 19078, 19089, 19102,
19135, 19164, 19169, 19174, 19195,
19204, 19213, 19218, 19225, 19238,
19240, 19242, 19244, 19250, 19272,
19286, 19314, 19319, 19353, 19388,
19409, 19411, 19421, 19430, 19435,
19444, 19453, 19469, 19482, 19488,
19499, 19512, 19518, 19537, 19557,
19587, 19598, 19613, 19626, 19644,
19652, 19657, 19659, 19661, 19678,
19696, 19698, 19721, 19733, 19753,
19774, 19781, 19788, 19800, 19806,
19863, 19899, 19908, 19927, 19946,
19952, 20015, 20025, 20027, 20029,
20036, 20092, 20105, 20121, 20132,
20151, 20166, 20173, 20180, 20182,
20196, 20203, 20217, 20233, 20242,
20256, 20271, 20289, 20298, 20300,

20312, 20324, 20336, 20349, 20356,
 20376, 20407, 20441, 20459, 20465,
 20474, 20519, 20535, 20552, 20574,
 20583, 20594, 20623, 20638, 20653,
 20662, 20674, 20681, 20683, 20685,
 20705, 20710, 20717, 20722, 20727,
 20732, 20781, 20819, 20861, 20877,
 20897, 20909, 20923, 20958, 20972,
 20983, 20991, 21000, 21039, 21045,
 21048, 21056, 21062, 21065, 21074,
 21077, 21080, 21083, 21088, 21097,
 21100, 21103, 21108, 21114, 21119,
 21124, 21129, 21137, 21157, 21159,
 21163, 21164, 21188, 21196, 21205,
 21217, 21226, 21234, 21274, 21318,
 21345, 21350, 21352, 21382, 21410,
 21722, 21724, 21726, 21737, 21763,
 21770, 21772, 21776, 21778, 21783,
 21786, 21791, 21794, 21808, 21810,
 21812, 21821, 21823, 21825, 21827,
 21845, 21848, 21857, 21860, 21863,
 21866, 21868, 21875, 21893, 21896,
 21902, 21910, 21916, 21924, 21931,
 21939, 21946, 21953, 21961, 21966,
 21971, 21973, 21979, 21981, 21983,
 21988, 21994, 22000, 22008, 22014,
 22022, 22028, 22036, 22043, 22051,
 22058, 22065, 22078, 22092, 22102,
 22136, 22147, 22158, 22169, 22180,
 22191, 22204, 22220, 22227, 22237,
 22242, 22257, 22272, 22284, 22300,
 22310, 22324, 22327, 22330, 22342,
 22381, 22385, 22389, 22392, 22442,
 22451, 22460, 22473, 22488, 22519,
 22533, 22535, 22566, 22568, 22589,
 22599, 22606, 22613, 22620, 22631,
 22642, 22650, 22671, 22705, 22720,
 22803, 22817, 22858, 22879, 22889,
 22908, 22913, 22927, 22932, 22945,
 22956, 22968, 22980, 23040, 23087,
 23095, 23124, 23173, 23181, 23205,
 23208, 23211, 23255, 23260, 23267,
 23270, 23272, 23274, 23276, 23278,
 23293, 23294, 23595, 23598, 23644,
 23714, 23749, 23767, 23773, 23779,
 23791, 23810, 23819, 23830, 23836,
 23841, 23849, 23862, 23875, 23891,
 23906, 23913, 23919, 23928, 23935,
 23944, 23946, 23948, 23950, 24001,
 24007, 24014, 24052, 24054, 24056,
 24076, 24147, 24149, 24151, 24157,
 24159, 24161, 24167, 24169, 24171,
 24177, 24179, 24187, 24198, 24199,
 24200, 24220, 24239, 24263, 24286,
 24293, 24299, 24306, 24361, 24363,
 24367, 24385, 24388, 24390, 24394,
 25378, 25381, 25383, 25385, 25397,
 25403, 25409, 25415, 25420, 25437,
 25438, 25441, 25445, 25448, 25459,
 25461, 25473, 25478, 25483, 25633,
 25650, 25655, 25662, 25667, 25669,
 25698, 25703, 25723, 25729, 25738,
 25740, 25745, 25747, 25756, 25761,
 25771, 25799, 25810, 25815, 25817,
 25819, 25829, 25831, 25851, 25854,
 25860, 25865, 25867, 25869, 25891,
 25893, 25908, 25923, 25927, 25933,
 25938, 25940, 25942, 25950, 25958,
 25967, 25977, 25979, 25982, 25984,
 25998, 26003, 26024, 26046, 26049,
 26062, 26075, 26080, 26082, 26084,
 26086, 26088, 26090, 26092, 26094,
 26103, 26112, 26114, 26116, 26121,
 26126, 26129, 26135, 26147, 26197,
 26212, 26214, 26240, 26253, 26268,
 26280, 26291, 26319, 26328, 26331,
 26333, 26335, 26350, 26352, 26365,
 26379, 26386, 26394, 26398, 26402,
 26405, 26420, 26430, 26465, 26472,
 26478, 26484, 26487, 26494, 26503,
 26508, 26516, 26527, 26529, 26533,
 26538, 26543, 26553, 26562, 26564,
 26567, 26569, 26571, 26573, 26578,
 26583, 26588, 26590, 26603, 26608,
 26610, 26612, 26614, 26616, 26618,
 26620, 26622, 26631, 26640, 26642,
 26644, 26649, 26661, 26676, 26701,
 26713, 26725, 26737, 26744, 26767,
 26770, 26772, 26774, 26777, 26830,
 26843, 26864, 26879, 26884, 26886,
 26894, 26904, 26912, 26917, 26922,
 26933, 26943, 26953, 26955, 26957,
 26959, 26990, 26992, 26997, 26999,
 27001, 27004, 27025, 27036, 27049,
 27051, 27053, 27055, 27057, 27059,
 27061, 27063, 27065, 27076, 27084,
 27087, 27097, 27113, 27128, 27150
 \cs_new_protected:Npx
 . . . 10, 336, 337, 341, 1770, 2161,
 2197, 2202, 2293, 2357, 2773, 2777,
 2782, 2954, 2958, 2959, 3048, 3997,
 7263, 7781, 8400, 8402, 8404, 8406,
 8408, 8417, 8419, 8421, 8681, 8683,
 8685, 8687, 8689, 8698, 8700, 8702,
 19382, 19396, 19398, 23581, 25833,
 25842, 26169, 26177, 26183, 26189
 \cs_new_protected_nopar:Nn
 12, 2291, 2355

`\cs_new_protected_nopar:Npn` 10, [2161](#), [2178](#), [2191](#), [2195](#)
`\cs_new_protected_nopar:Npx` 10, [2161](#), [2191](#), [2196](#)
`\cs_set:Nn` 12, [321](#), [2291](#), [2355](#)
`\cs_set:Npn` 9,
10, [92](#), [92](#), [291](#), [317](#), [321](#), [490](#), [1459](#),
1489, 1496, 1498, 1499, 1500, 1501,
1502, 1503, 1504, 1505, 1506, 1507,
1508, 1509, 1510, 1511, 1512, 1513,
1514, 1515, 1516, 1517, 1519, 1520,
1521, 1522, 1523, 1524, 1525, 1526,
1527, 1583, 1680, 1689, 1691, 1789,
1823, 1829, 1842, 1844, 2008, 2012,
2016, 2020, 2025, 2031, 2032, 2036,
2043, 2046, 2106, 2108, 2110, 2112,
2114, 2116, 2118, 2120, 2139, 2161,
2177, [2185](#), 2185, 2291, 2355, 3311,
3376, 3483, 3653, 4045, 4081, 5286,
5299, 6831, 6920, 7313, 7773, 7779,
7965, 8220, 8222, 9376, 9675, 10738,
11952, 12014, 12022, 12031, 12048,
12056, 12084, 17752, 18790, 18791,
18792, 19097, 19098, 19665, 19667,
19683, 19685, 19979, 20006, 20045,
20555, 20863, 23307, 24262, 25046
`\cs_set:Npx` . . . 10, [326](#), [574](#), [1459](#),
[2185](#), 2186, 3378, 6907, 7696, 7702,
9423, 9424, 9425, 9426, 9427, 10723,
10731, 10994, 18532, 18538, 20219
`\cs_set_eq:NN` 14, [93](#), [302](#), [318](#), [437](#),
1572, 1580, 1960, [2203](#), 2777, 2795,
2958, 3094, 3104, 4079, 4080, 4903,
4904, 4906, 5013, 5014, 5025, 6171,
6245, 6248, 7266, 7459, 7689, 7694,
7701, 7784, 9429, 9430, 9431, 9433,
9435, 10989, 11004, 11008, 11036,
11047, 11057, 18002, 18003, 18008,
18163, 18206, 18231, 19971, 19980,
20003, 20561, 23461, 24209, 24437
`\cs_set_nopar:Nn` 12, [2291](#), [2355](#)
`\cs_set_nopar:Npn` 9,
10, [118](#), [301](#), [317](#), [1459](#), 1488, 1568,
1729, 1730, [2177](#), 2179, 10946, 11015
`\cs_set_nopar:Npx`
. 10, [1004](#), [1459](#), 1492, [2177](#),
2180, 2482, 2685, 3142, 3145, 3148,
3166, 3169, 3172, 3175, 3198, 3201,
3204, 3207, 25379, 25399, 25405, 25442
`\cs_set_protected:Nn` . 12, [2291](#), [2355](#)
`\cs_set_protected:Npn`
. 9, 10, 253, [318](#),
[1459](#), 1475, 1477, 1479, 1481, 1483,
1485, 1490, 1529, 1532, 1537, 1545,
1555, 1560, 1569, 1574, 1586, 1587,
1591, 1593, 1602, 1611, 1616, 1622,
1629, 1631, 1632, 1633, 1634, 1635,
1637, 1648, 1650, 1662, 1678, 1688,
1709, 1711, 1714, 1715, 1725, 1727,
1731, 1738, 1742, 1758, 1768, 1779,
1794, 1797, 1799, 1801, 1803, 1807,
1808, 1810, 1812, 1816, 1818, 1821,
1827, 1833, 1837, 1838, 1846, 1848,
1850, 1852, 1854, 1863, 1865, 1867,
1869, 1871, 1880, 1892, 1907, 1924,
1941, 1947, 1953, 1959, 1961, 1963,
1975, 1989, 2034, 2122, 2135, 2137,
2141, 2143, 2150, 2158, 2163, [2197](#),
2197, 2231, 2252, 3490, 3966, 3993,
4930, 5425, 7255, 7379, 7547, 7565,
8323, 8386, 8668, 8670, 9032, 9356,
9358, 9477, 9557, 9671, 9864, 10029,
10418, 10460, 10554, 10983, 11969,
12454, 12531, 13115, 13189, 13203,
13398, 13415, 13450, 13484, 13499,
13516, 15098, 18004, 19394, 19419,
19428, 19956, 19965, 19967, 19969,
19972, 19974, 19981, 19983, 19988,
19990, 19995, 19997, 19999, 20001,
20004, 20724, 20725, 21161, 22524,
22545, 23316, 23463, 23466, 23486,
23503, 23523, 23531, 23564, 23961,
23975, 23986, 25087, 25094, 25097,
25123, 25134, 25265, 25268, 25290,
25631, 25708, 25766, 27236, 27240
`\cs_set_protected:Npx`
. 10, 239, [1459](#), [2197](#), 2198, 8543
`\cs_set_protected_nopar:Nn`
. 13, [2291](#), [2355](#)
`\cs_set_protected_nopar:Npn`
. 11, [318](#), [1459](#), [2191](#), 2191
`\cs_set_protected_nopar:Npx`
. 11, [1459](#), [2191](#), 2192
`\cs_show:N` 15, 15, 20, [324](#), [2432](#)
`\cs_split_function:N`
. 16, 1859, 1876, 1968,
1969, [2034](#), 2263, 2304, 2764, 2987
`\cs_to_sr:N` 1007
`\cs_to_str:N` . . . 4, 16, [41](#), [48](#), [312](#),
[312](#), [313](#), 388, 1649, [2025](#), 2040,
4576, 4577, 4578, 4579, 4580, 4581,
4582, 4583, 4584, 4585, 4586, 4587,
9361, 11967, 13747, 18546, 19937,
25373, 25384, 25476, 25481, 25489
`\cs_undefine:N` 14, [434](#), 576,
[2219](#), 8706, 8707, 8708, 23289, 23290
cs internal commands:
`__cs_args_generate:n` [3028](#)

<code>__cs_args_generate:Nn</code>	3028	<code>__cs_generate_variant_TF_-</code>	
<code>__cs_count_signature:N</code> . . .	311 , 2262	<code>form:nnn</code>	2982
<code>__cs_count_signature:n</code>	2262	<code>__cs_get_function_name:N</code>	311
<code>__cs_count_signature:nnN</code>	2262	<code>__cs_get_function_signature:N</code> .	311
<code>__cs_generate_from_signature:n</code> .		<code>__cs_parm_from_arg_count_-</code>	
	2313 , 2326	<code>test:nnTF</code>	2231
<code>__cs_generate_from_signature:NNn</code>		<code>__cs_split_function_auxi:w</code> ..	2034
	2295 , 2299	<code>__cs_split_function_auxii:w</code> .	2034
<code>__cs_generate_from_signature:nnNNNn</code>		<code>__cs_tmp:w</code>	312 ,
	2303 , 2308	336 , 341 , 2034 , 2049 , 2161 , 2177 ,	
<code>__cs_generate_internal_variant:n</code>		2179 , 2180 , 2181 , 2182 , 2183 , 2184 ,	
	2949 , 2954	2185 , 2186 , 2187 , 2188 , 2189 , 2190 ,	
<code>__cs_generate_internal_variant:wnNwnn</code>		2191 , 2192 , 2193 , 2194 , 2195 , 2196 ,	
	2956 , 2971	2197 , 2198 , 2199 , 2200 , 2201 , 2202 ,	
<code>__cs_generate_internal_variant:wnnw</code>		2291 , 2331 , 2332 , 2333 , 2334 , 2335 ,	
	2954	2336 , 2337 , 2338 , 2339 , 2340 , 2341 ,	
<code>__cs_generate_internal_variant_-</code>		2342 , 2343 , 2344 , 2345 , 2346 , 2347 ,	
<code>loop:n</code>	2954	2348 , 2349 , 2350 , 2351 , 2352 , 2353 ,	
<code>__cs_generate_variant:N</code> .	2760 , 2773	2354 , 2355 , 2363 , 2364 , 2365 , 2366 ,	
<code>__cs_generate_variant:n</code>	2982	2367 , 2368 , 2369 , 2370 , 2371 , 2372 ,	
<code>__cs_generate_variant:nnNN</code>		2373 , 2374 , 2375 , 2376 , 2377 , 2378 ,	
	2763 , 2796	2379 , 2380 , 2381 , 2382 , 2383 , 2384 ,	
<code>__cs_generate_variant:nnNnn</code> .	2982	2385 , 2386 , 2777 , 2795 , 2950 , 2958	
<code>__cs_generate_variant:Nnnw</code>		<code>__cs_to_str:N</code>	312 , 2025
	2803 , 2805	<code>__cs_to_str:w</code>	312 , 2025
<code>__cs_generate_variant:w</code>	2982	<code>csc</code>	192
<code>__cs_generate_variant:ww</code>	2773	<code>cscd</code>	193
<code>__cs_generate_variant:wwNN</code>		<code>\csname</code>	14 , 21 , 39 , 43 , 49 ,
	338 , 339 , 2812 , 2930	68 , 90 , 92 , 98 , 123 , 146 , 150 , 221 , 320	
<code>__cs_generate_variant:wwNw</code> ..	2773	<code>\currentgrouplevel</code>	612
<code>__cs_generate_variant_F_-</code>		<code>\currentgrouptype</code>	613
<code>form:nnn</code>	2982	<code>\currentifbranch</code>	614
<code>__cs_generate_variant_loop:nNwN</code>		<code>\currentiflevel</code>	615
	338 , 339 , 2813 , 2825	<code>\currentiftype</code>	616
<code>__cs_generate_variant_loop_-</code>			
<code>base:N</code>	2825		
<code>__cs_generate_variant_loop_-</code>			
<code>end:nwwwNNnn</code> .	338 , 339 , 2815 , 2825		
<code>__cs_generate_variant_loop_-</code>			
<code>invalid:NNwNNnn</code>	339 , 2825		
<code>__cs_generate_variant_loop_-</code>			
<code>long:wNNnn</code>	339 , 2818 , 2825		
<code>__cs_generate_variant_loop_-</code>			
<code>same:w</code>	339 , 2825		
<code>__cs_generate_variant_loop_-</code>			
<code>special:NNwNNnn</code>	2825 , 2925		
<code>__cs_generate_variant_loop_-</code>			
<code>warning:nnnnnn</code>	2825 , 27234		
<code>__cs_generate_variant_p_-</code>			
<code>form:nnn</code>	2982		
<code>__cs_generate_variant_same:N</code> . . .			
	338 , 2868 , 2919		
<code>__cs_generate_variant_T_-</code>			
<code>form:nnn</code>	2982		
<code>__cs_generate_variant_TF_-</code>			
<code>form:nnn</code>	2982		

D

<code>\day</code>	321
<code>dd</code>	195
<code>\deadcycles</code>	322
debug commands:	
<code>\debug_deprecation_off:</code>	433
<code>\debug_deprecation_on:</code>	433 , 544
<code>\debug_off:n</code>	235 , 446 , 1535
<code>\debug_on:n</code>	235 , 446 , 1535
<code>\debug_resume:</code>	235 , 299 , 930 , 1566 , 22470
<code>\debug_suspend:</code>	235 , 299 , 930 , 1566 , 22463
debug internal commands:	
<code>__debug_check-declarations_off:</code>	1589
<code>__debug_check-declarations_on:</code>	1589
<code>__debug_check-expressions_off:</code>	1676
<code>__debug_check-expressions_on:</code>	1676

- __debug_chk_expr_aux:nNnN . . . [1676](#)
- __debug_chk_var_scope_aux:NN . . .
 [1614](#), [1620](#), [1626](#), [1646](#)
- __debug_chk_var_scope_aux:Nn . [1646](#)
- __debug_chk_var_scope_aux:NNn . .
 [301](#), [1646](#)
- __debug_deprecation_aux:nnNnn [1740](#)
- __debug_deprecation_expandable:nnNnn
 [1756](#), [1779](#)
- __debug_deprecation_off: [1723](#)
- \g__debug_deprecation_off_tl . . .
 [303](#), [1723](#), [1765](#), [1786](#)
- __debug_deprecation_on: [1723](#)
- \g__debug_deprecation_on_tl
 [303](#), [1723](#), [1760](#), [1781](#)
- __debug_log-functions_off: . . [1707](#)
- __debug_log-functions_on: . . [1707](#)
- __debug_patch_args_aux:nnnn . [1810](#)
- __debug_patch_args_aux:nnnNNnn [1810](#)
- __debug_patch_args_aux:nnnNNnnn
 [1810](#)
- __debug_patch_aux:nnnn [1795](#)
- __debug_patch_auxii:nnnn [1795](#)
- __debug_suspended:TF
 [299](#), [1566](#), [1595](#),
 [1604](#), [1613](#), [1618](#), [1624](#), [1682](#), [1712](#)
- \l__debug_suspended_tl [1566](#)
- __debug_tmp:w [1810](#)
- \def [74](#),
 [75](#), [76](#), [105](#), [122](#), [124](#), [125](#), [143](#), [144](#),
 [147](#), [163](#), [178](#), [206](#), [210](#), [235](#), [274](#), [323](#)
- default commands:
 .default:n [168](#), [11135](#)
- \defaultthyphenchar [324](#)
- \defaultskewchar [325](#)
- deg [195](#)
- \delcode [326](#)
- \delimiter [327](#)
- \delimiterfactor [328](#)
- \delimitershortfall [329](#)
- \detokenize [68](#), [221](#), [617](#)
- \DH [25302](#)
- \dh [25302](#)
- dim commands:
 \dim_abs:n [151](#), [10075](#)
- \dim_add:Nn [151](#), [10053](#)
- \dim_case:nn [154](#), [10173](#)
- \dim_case:nnn [27168](#)
- \dim_case:nnTF
 [154](#), [10173](#), [10178](#), [10183](#), [27169](#)
- \dim_compare:nNnTF [152](#),
 [154](#), [154](#), [154](#), [155](#), [10122](#), [10197](#),
 [10233](#), [10241](#), [10250](#), [10256](#), [10283](#),
 [10286](#), [10297](#), [22723](#), [22726](#), [22729](#),
 [22738](#), [22741](#), [22744](#), [22753](#), [22760](#),
 [22823](#), [22828](#), [22839](#), [22958](#), [22970](#),
 [23606](#), [23623](#), [23652](#), [23666](#), [23676](#)
- \dim_compare:nTF [153](#), [155](#), [155](#), [155](#),
 [155](#), [10133](#), [10205](#), [10213](#), [10222](#), [10228](#)
- \dim_compare_p:n [153](#), [10133](#)
- \dim_compare_p:nNn [152](#), [10122](#)
- \dim_const:Nn [150](#),
 [546](#), [557](#), [10005](#), [10380](#), [10381](#), [17562](#)
- \dim_do_until:nn [155](#), [10203](#)
- \dim_do_until:nNnn [154](#), [10231](#)
- \dim_do_while:nn [155](#), [10203](#)
- \dim_do_while:nNnn [154](#), [10231](#)
- \dim_eval:n
 [152](#), [153](#), [156](#), [156](#), [546](#), [910](#),
 [10009](#), [10176](#), [10181](#), [10186](#), [10191](#),
 [10301](#), [10329](#), [10375](#), [10379](#), [22509](#),
 [22556](#), [22626](#), [22637](#), [22654](#), [22658](#),
 [22659](#), [22663](#), [22667](#), [22668](#), [22675](#),
 [22680](#), [22686](#), [22693](#), [22700](#), [22941](#),
 [22942](#), [23218](#), [23219](#), [23220](#), [23731](#),
 [23752](#), [23755](#), [23756](#), [23763](#), [23845](#),
 [23846](#), [23853](#), [23854](#), [23932](#), [23939](#)
- \dim_gadd:Nn [151](#), [10053](#)
- .dim_gset:N [168](#), [11143](#)
- \dim_gset:Nn [151](#), [546](#), [10037](#)
- \dim_gset_eq:NN [151](#), [10045](#)
- \dim_gsub:Nn [151](#), [10053](#)
- \dim_gzero:N [150](#), [10012](#), [10022](#)
- \dim_gzero_new:N [150](#), [10019](#)
- \dim_if_exist:NnTF
 [150](#), [10020](#), [10022](#), [10025](#)
- \dim_if_exist_p:N [150](#), [10025](#)
- \dim_log:N [158](#), [10376](#)
- \dim_log:n [158](#), [10376](#)
- \dim_max:nn [151](#), [10075](#), [23824](#), [23828](#)
- \dim_min:nn
 [151](#), [10075](#), [23822](#), [23826](#), [23839](#)
- \dim_new:N
 [150](#), [150](#), [9997](#), [10008](#), [10020](#),
 [10022](#), [10382](#), [10383](#), [10384](#), [10385](#),
 [22083](#), [22084](#), [22085](#), [22086](#), [22087](#),
 [22088](#), [22089](#), [22090](#), [22397](#), [22421](#),
 [22422](#), [22425](#), [22426](#), [22427](#), [22428](#),
 [23028](#), [23030](#), [23031](#), [23709](#), [23710](#),
 [23711](#), [23712](#), [23713](#), [23860](#), [23861](#)
- \dim_ratio:nn [152](#), [555](#), [10118](#), [10369](#)
- .dim_set:N [168](#), [11143](#)
- \dim_set:Nn [151](#), [10037](#), [22104](#),
 [22105](#), [22106](#), [22138](#), [22149](#), [22222](#),
 [22223](#), [22224](#), [22239](#), [22312](#), [22313](#),
 [22314](#), [22316](#), [22318](#), [22320](#), [22494](#),
 [22540](#), [22725](#), [22730](#), [22740](#), [22745](#),
 [22755](#), [22762](#), [22775](#), [22806](#), [22826](#),

- 22892, 22893, 22895, 22897, 22915,
 22916, 23029, 23132, 23133, 23184,
 23185, 23186, 23188, 23758, 23793,
 23801, 23812, 23813, 23814, 23815,
 23821, 23823, 23825, 23827, 23832,
 23838, 23896, 23898, 23900, 23908,
 23910, 26313, 26314, 26511, 26512
 \dim_set_eq:NN
 151, 10045, 22496, 22497, 22542, 22543
 \dim_show:N 157, 10372
 \dim_show:n 158, 556, 10374
 \dim_step_function:nnnN
 155, 554, 10259, 10326
 \dim_step_inline:nnnn ... 155, 10304
 \dim_step_variable:nnnNn . 156, 10304
 \dim_sub:Nn 151, 10053
 \dim_to_decimal:n 156, 10336, 10354,
 10366, 26788, 26789, 26790, 26791,
 26792, 26794, 26869, 26870, 26915,
 26920, 26926, 26927, 26928, 26929,
 26938, 26939, 26940, 27031, 27050
 \dim_to_decimal_in_bp:n
 157, 157, 10353, 25880,
 25881, 25882, 25946, 25947, 25954,
 25955, 25962, 25963, 25971, 25972,
 25973, 26070, 26074, 26078, 26203,
 26204, 26205, 26341, 26342, 26343,
 26444, 26445, 26446, 26447, 26536,
 26541, 26547, 26548, 26549, 26557,
 26558, 26598, 26602, 26606, 27035
 \dim_to_decimal_in_sp:n 157,
 157, 632, 10355, 12594, 12631, 13228
 \dim_to_decimal_in_unit:nn 157, 10364
 \dim_to_fp:n
 157, 632, 651, 10372, 17273,
 22142, 22143, 22153, 22154, 22210,
 22213, 22214, 22240, 22249, 22250,
 22264, 22265, 22278, 22290, 22293,
 22294, 22768, 22770, 22780, 22782,
 22784, 22785, 22810, 22811, 22812,
 22813, 23797, 23798, 23805, 23806,
 23865, 23868, 23869, 23909, 23911
 \dim_until_do:nn 155, 10203
 \dim_until_do:nNnn 154, 10231
 \dim_use:N
 156, 156, 910, 10080, 10091,
 10092, 10093, 10104, 10105, 10106,
 10136, 10155, 10332, 10333, 10341,
 22923, 23760, 23764, 23771, 23777,
 23786, 23787, 23788, 23917, 23924
 \dim_while_do:nn 155, 10203
 \dim_while_do:nNnn 155, 10231
 \dim_zero:N 150, 150, 10012, 10020,
 22107, 22225, 22315, 22716, 22717
 \dim_zero_new:N 150, 10019
 \c_max_dim 158, 161,
 10380, 10516, 17609, 17620, 17622,
 23812, 23813, 23814, 23815, 23832
 \g_tmpa_dim 158, 10382
 \l_tmpa_dim 158, 10382
 \g_tmpb_dim 158, 10382
 \l_tmpb_dim 158, 10382
 \c_zero_dim 158, 10283,
 10286, 10380, 10515, 17573, 17574,
 21968, 21990, 22373, 22723, 22726,
 22729, 22738, 22741, 22744, 22753,
 22760, 22823, 22828, 22839, 23610,
 23621, 23627, 23639, 23652, 23656,
 23664, 23666, 23670, 23676, 23686
 dim internal commands:
 __dim_abs:N 10075
 __dim_case:nnTF 10173
 __dim_case:nw 10173
 __dim_case_end:nw 10173
 __dim_compare:w 10133
 __dim_compare:wNN 550, 10133
 __dim_compare!:w 10133
 __dim_compare<:w 10133
 __dim_compare=:w 10133
 __dim_compare>:w 10133
 __dim_compare_end:w .. 10141, 10165
 __dim_compare_error: ... 550, 10133
 __dim_eval:w
 555, 9994, 10034, 10039, 10042,
 10055, 10060, 10066, 10071, 10076,
 10080, 10086, 10087, 10091, 10092,
 10093, 10099, 10100, 10104, 10105,
 10106, 10121, 10124, 10126, 10130,
 10136, 10155, 10160, 10262, 10266,
 10270, 10277, 10278, 10279, 10330,
 10332, 10337, 10341, 10358, 10363
 __dim_eval_end: 9994,
 10039, 10042, 10055, 10060, 10066,
 10071, 10080, 10095, 10108, 10121,
 10125, 10130, 10332, 10341, 10363
 __dim_maxmin:wwN 10075
 __dim_ratio:n 10118
 __dim_step:NnnN 10259
 __dim_step:NNnnnn 10304
 __dim_step:wwwN 10259
 __dim_tmp:w 547, 548, 10029, 10037,
 10040, 10053, 10056, 10064, 10067
 __dim_to_decimal:w 10336
 \dimen 330, 7606
 \dimendef 331
 \dimexpr 618
 \directlua 16, 23, 53, 59, 61, 897
 \disablecjktoken 1227

- \discretionary 332
- \displayindent 333
- \displaylimits 334
- \displaystyle 335
- \displaywidowpenalties 619
- \displaywidowpenalty 336
- \displaywidth 337
- \divide 338
- \DJ 25303
- \dj 25303
- \do 1266
- \doublehyphenemerits 339
- \dp 340
- \draftmode 992
- driver commands:
 - \driver_box_use_clip:N
252, 23596, 25869, 26197, 26335, 26777
 - \driver_box_use_rotate:Nn
253, 22125, 25891, 26212, 26350, 26830
 - \driver_box_use_scale:Nnn
253, 22346, 25908, 26240, 26365, 26843
 - \driver_color_cmyk:nnnn
.... 253, 23273, 25729, 25799, 27075
 - \driver_color_gray:n
.... 253, 23275, 25729, 25799, 27086
 - \driver_color_pickup:N
.... 253, 23263, 25702, 25760
 - \driver_color_rgb:nnn
.... 253, 23277, 25729, 25799, 27112
 - \driver_color_spot:nn
.... 23279, 25729, 25799
 - \driver_draw_begin:
.... 254, 25927, 26527, 26879
 - \driver_draw_box:Nnnnnnn 257
 - \driver_draw_box_use:Nnnnn
.... 257, 1019, 26147, 26744, 27128
 - \driver_draw_cap_but:
.... 256, 256, 26062, 26590, 27025
 - \driver_draw_cap_rectangle:
.... 256, 26062, 26590, 27025
 - \driver_draw_cap_round:
.... 256, 26062, 26590, 27025
 - \driver_draw_clip:
.... 255, 25982, 26567, 26957
 - \driver_draw_closepath:
.... 254, 25982, 26567, 26957
 - \driver_draw_closestroke:
.... 255, 25982, 26567, 26957
 - \driver_draw_cm:nnnn
.... 257, 26135, 26155, 26156,
26157, 26661, 26748, 27113, 27131
 - \driver_draw_color_fill_cmyk:nnnn
.... 256, 26094, 26622, 27065
 - \driver_draw_color_fill_gray:n ..
.... 256, 26094, 26622, 27065
 - \driver_draw_color_fill_rgb:nnn ..
.... 256, 26094, 26622, 27065
 - \driver_draw_color_gray:n 27076
 - \driver_draw_color_rgb:nnn ... 27087
 - \driver_draw_color_stroke_-
cmyk:nnnn 256, 26094, 26622, 27065
 - \driver_draw_color_stroke_gray:n
.... 256, 26094, 26622, 27065
 - \driver_draw_color_stroke_-
rgb:nnn .. 256, 26094, 26622, 27065
 - \driver_draw_curveto:nnnnnn
.... 254, 25942, 26533, 26912
 - \driver_draw_dash:nn 256
 - \driver_draw_dash_pattern:nn
.... 256, 26062, 26590, 27025
 - \driver_draw_discardpath:
.... 255, 25982, 26567, 26957
 - \driver_draw_end:
.... 254, 25927, 26527, 26879
 - \driver_draw_evenodd_rule:
.... 255, 25977, 26562, 26953
 - \driver_draw_fill:
.... 255, 25982, 26567, 26957
 - \driver_draw_fillstroke:
.... 255, 25982, 26567, 26957
 - \driver_draw_join_bevel:
.... 256, 26062, 26590, 27025
 - \driver_draw_join_miter:
.... 256, 26062, 26590, 27025
 - \driver_draw_join_round:
.... 256, 26062, 26590, 27025
 - \driver_draw_lineto:nn
.... 254, 25942, 26533, 26912
 - \driver_draw_linewidth:n
.... 256, 26062, 26590, 27025
 - \driver_draw_miterlimit:n
.... 256, 26062, 26590, 27025
 - \driver_draw_move:nn 254
 - \driver_draw_moveto:nn
.... 254, 25942, 26533, 26912
 - \driver_draw_nonzero_rule:
.... 255, 25977, 26562, 26953
 - \driver_draw_rectangle:nnnn
.... 254, 25942, 26533, 26912
 - \driver_draw_scope_begin:
254, 25938, 26528, 26531, 26881, 26886
 - \driver_draw_scope_end:
254, 25938, 26530, 26531, 26885, 26886
 - \driver_draw_stroke:
.... 255, 255, 25982, 26567, 26957
 - \driver_draw_stroke_cmyk:nnnn .. 253
 - \driver_draw_stroke_gray:n 253

- \driver_draw_stroke_rgb:nnn ... [253](#)
 - driver internal commands:
 - __driver_align_currentpoint...
 - [1024](#)
 - __driver_align_currentpoint_-
 - begin: .. [25854](#), [25872](#), [25896](#), [25911](#)
 - __driver_align_currentpoint_-
 - end: [25854](#), [25886](#), [25904](#), [25918](#)
 - __driver_box_use_rotate:Nn
 - [25891](#), [26212](#), [26350](#)
 - \g_driver_clip_path_int
 - [26777](#), [26963](#),
[26966](#), [26979](#), [27008](#), [27011](#), [27019](#)
 - __driver_color_cmyk:nnnn [25799](#)
 - __driver_color_fill_select:n . [26622](#)
 - __driver_color_gray:n [25799](#)
 - __driver_color_gray:n . [25816](#), [25817](#)
 - __driver_color_pickup:w
 - [1014](#), [25702](#), [25760](#)
 - __driver_color_reset:
 - [25729](#), [25799](#), [26132](#)
 - __driver_color_rgb:nnn [25799](#)
 - __driver_color_select:n . [25729](#),
[25799](#), [26633](#), [26643](#), [26651](#), [26655](#)
 - \l_driver_color_stack_int
 - [25798](#), [25838](#), [25847](#)
 - \l_driver_cos_fp [26212](#)
 - __driver_draw_add_to_path:n ...
 - [26912](#), [26958](#)
 - \g_driver_draw_clip_bool
 - [25982](#), [26957](#)
 - __driver_draw_cm:nnnn [26661](#)
 - __driver_draw_cm_decompose:nnnnN
 - [26671](#), [26700](#)
 - __driver_draw_cm_decompose_-
 - auxi:nnnnN [26700](#)
 - __driver_draw_cm_decompose_-
 - auxii:nnnnN [26700](#)
 - __driver_draw_cm_decompose_-
 - auxiii:nnnnN [26700](#)
 - __driver_draw_color_fill:n . [26094](#)
 - __driver_draw_color_fill:nnn . [27065](#)
 - __driver_draw_color_gray_aux:n .
 - [27080](#), [27084](#)
 - __driver_draw_color_stroke:n . [26094](#)
 - __driver_draw_dash:n
 - [26062](#), [26590](#), [27025](#)
 - __driver_draw_dash_aux:nn ... [27025](#)
 - \g_driver_draw_eor_bool . [25977](#),
[25991](#), [26009](#), [26017](#), [26030](#), [26039](#),
[26055](#), [26562](#), [26576](#), [26581](#), [26586](#)
 - __driver_draw_literal:n . [25925](#),
[25930](#), [25931](#), [25935](#), [25939](#), [25941](#),
[25944](#), [25952](#), [25960](#), [25969](#), [25983](#),
[25986](#), [25989](#), [25995](#), [26005](#), [26006](#),
[26007](#), [26012](#), [26015](#), [26021](#), [26026](#),
[26027](#), [26028](#), [26033](#), [26034](#), [26037](#),
[26043](#), [26053](#), [26059](#), [26064](#), [26077](#),
[26081](#), [26083](#), [26085](#), [26087](#), [26089](#),
[26091](#), [26093](#), [26127](#), [26137](#), [26149](#),
[26150](#), [26151](#), [26152](#), [26153](#), [26154](#),
[26158](#), [26159](#), [26161](#), [26162](#), [26163](#),
[26164](#), [26165](#), [26525](#), [26535](#), [26540](#),
[26545](#), [26555](#), [26568](#), [26570](#), [26572](#),
[26575](#), [26580](#), [26585](#), [26589](#), [26592](#),
[26605](#), [26609](#), [26611](#), [26613](#), [26615](#),
[26617](#), [26619](#), [26621](#), [26658](#), [26877](#),
[26898](#), [26906](#), [26964](#), [26983](#), [27009](#)
- __driver_draw_path:n [26957](#)
- \g_driver_draw_path_int [26957](#)
- \g_driver_draw_path_tl . [26912](#),
[26968](#), [26984](#), [26986](#), [27013](#), [27022](#)
- __driver_draw_scope:n
 - [26882](#), [26886](#),
[26954](#), [26956](#), [26976](#), [27016](#), [27038](#),
[27050](#), [27052](#), [27054](#), [27056](#), [27058](#),
[27060](#), [27062](#), [27064](#), [27099](#), [27115](#)
- \g_driver_draw_scope_int [26886](#)
- \l_driver_draw_scope_int [26886](#)
- \l_driver_image_attr_tl
 - [26252](#), [26257](#), [26264](#),
[26272](#), [26282](#), [26315](#), [26317](#), [26322](#)
- __driver_image_getbb_aux:nNnn [26485](#)
- __driver_image_getbb_auxi:n . [26253](#)
- __driver_image_getbb_auxii:n . [26465](#)
- __driver_image_getbb_auxiii:n . [26253](#)
- __driver_image_getbb_auxiiii:nNnn .
 - [26465](#)
- __driver_image_getbb_auxiiiii:nNnn
 - [26465](#)
- __driver_image_getbb_auxiv:nnNnn
 - [26465](#), [26521](#)
- __driver_image_getbb_auxv:nnNnn .
 - [26465](#)
- __driver_image_getbb_auxvi:nnNnn
 - [26506](#), [26508](#)
- __driver_image_getbb_eps:n
 - [25922](#), [26377](#)
- __driver_image_getbb_jpg:n
 - [26253](#), [26377](#), [26465](#), [26862](#)
- __driver_image_getbb_pagebox:w .
 - [26465](#)
- __driver_image_getbb_pdf:n
 - [26253](#), [26377](#), [26465](#)
- __driver_image_getbb_png:n
 - [26253](#), [26377](#), [26465](#), [26862](#)
- __driver_image_include_auxi:nn .
 - [26394](#)

__driver_image_include_auxii:nnn	\dvifedback	899
..... 26394	\dvivariable	900
__driver_image_include_auxiii:nn		
..... 26394		
__driver_image_include_auxiii:nnn	E	
..... 26430	\edef	4 , 106 , 131 , 208 , 342
__driver_image_include_bitmap_-	\efcode	787
quote:w	\else	15 , 22 , 44 , 46 ,
__driver_image_include_eps:n		91 , 94 , 95 , 99 , 100 , 161 , 165 , 180 , 343
..... 25923 , 26394	else commands:	
__driver_image_include_jpg:n	\else:	20 , 88 , 89 , 89 , 93 , 99 , 99 ,
..... 26319 , 26394 , 26864		148 , 164 , 224 , 224 , 225 , 306 , 313 ,
__driver_image_include_pdf:n		336 , 356 , 368 , 369 , 677 , 1402 , 1446 ,
..... 26319 , 26394 , 26516		1641 , 1656 , 1665 , 1668 , 1718 , 1746 ,
__driver_image_include_png:n		1749 , 1929 , 2054 , 2057 , 2066 , 2072 ,
..... 26319 , 26394 , 26864		2082 , 2085 , 2094 , 2100 , 2225 , 2247 ,
\g_driver_image_int		2256 , 2270 , 2328 , 2329 , 2390 , 2504 ,
..... 26393 , 26432 , 26433		2750 , 2778 , 2829 , 2830 , 2832 , 2836 ,
__driver_literal:n		2848 , 2849 , 2850 , 2851 , 2852 , 2853 ,
25697 , 25749 ,		2854 , 2855 , 2921 , 2922 , 2924 , 3009 ,
25757 , 25852 , 25856 , 25863 , 25866 ,		3422 , 3433 , 3444 , 3454 , 3468 , 3514 ,
25868 , 25924 , 25929 , 25936 , 26131 ,		3780 , 3809 , 3830 , 3846 , 3854 , 3864 ,
26329 , 26332 , 26334 , 26355 , 26368 ,		3877 , 3893 , 4103 , 4110 , 4116 , 4352 ,
26396 , 26424 , 26434 , 26678 , 26685 ,		4408 , 4411 , 4414 , 4426 , 4441 , 4620 ,
26691 , 26751 , 26761 , 26768 , 26866		4658 , 4666 , 4691 , 4919 , 4950 , 4959 ,
__driver_literal_pdf:n		5280 , 5311 , 5332 , 5335 , 5399 , 5521 ,
.. 26169 , 26200 , 26328 , 26338 , 26525		5554 , 5594 , 5604 , 5920 , 5953 , 6004 ,
__driver_literal_postscript:n		6142 , 6174 , 6215 , 6225 , 6275 , 6307 ,
..... 25751 , 25851 , 25857 ,		6324 , 6346 , 6483 , 6485 , 6487 , 6489 ,
25858 , 25862 , 25873 , 25874 , 25876 ,		6726 , 6741 , 6763 , 6777 , 7285 , 7288 ,
25877 , 25885 , 25897 , 25912 , 25925		7296 , 7302 , 7341 , 7347 , 7441 , 7446 ,
__driver_literal_svg:n		7451 , 7456 , 7463 , 7470 , 7475 , 7480 ,
26767 , 26771 , 26773 , 26775 , 26780 ,		7485 , 7490 , 7495 , 7500 , 7505 , 7510 ,
26782 , 26799 , 26877 , 27132 , 27143		7532 , 7538 , 7541 , 7576 , 7579 , 7643 ,
__driver_literal_x:n		7652 , 7660 , 7669 , 7725 , 7739 , 7748 ,
..... 25697		7758 , 7768 , 8119 , 8939 , 8942 , 8943 ,
__driver_matrix:n		9186 , 10083 , 10114 , 10131 , 10141 ,
..... 26189 , 26222 , 26243 , 26664		10166 , 10660 , 10684 , 10702 , 10713 ,
\g_driver_path_int		10725 , 10737 , 10753 , 11624 , 11634 ,
... 26972 , 26989		11635 , 11650 , 11660 , 11755 , 11831 ,
__driver_scope_begin:		11893 , 11896 , 11910 , 11928 , 11932 ,
1041 , 25865 ,		12197 , 12210 , 12230 , 12258 , 12259 ,
25871 , 25895 , 25910 , 26177 , 26199 ,		12281 , 12302 , 12325 , 12326 , 12368 ,
26216 , 26242 , 26331 , 26337 , 26354 ,		12386 , 12421 , 12425 , 12460 , 12477 ,
26367 , 26531 , 26746 , 26770 , 27130		12483 , 12487 , 12491 , 12652 , 12685 ,
__driver_scope_begin:n		12693 , 12726 , 12730 , 12742 , 12752 ,
.. 26774 ,		12762 , 12793 , 12806 , 12841 , 12851 ,
26801 , 26809 , 26814 , 26832 , 26845		12870 , 12883 , 12896 , 12900 , 12911 ,
__driver_scope_end:		12934 , 12951 , 12963 , 12977 , 12993 ,
.... 25865 ,		13001 , 13003 , 13013 , 13024 , 13040 ,
25888 , 25906 , 25920 , 26177 , 26209 ,		13056 , 13062 , 13067 , 13074 , 13096 ,
26236 , 26250 , 26331 , 26347 , 26363 ,		13126 , 13149 , 13177 , 13180 , 13356 ,
26375 , 26532 , 26763 , 26770 , 26824 ,		13360 , 13367 , 13387 , 13406 , 13423 ,
26825 , 26826 , 26841 , 26860 , 27144		13429 , 13461 , 13491 , 13507 , 13527 ,
\l_driver_sin_fp		13568 , 13583 , 13616 , 13618 , 13624 ,
..... 26212		
\l_driver_tmp_box		
..... 26311 ,		
26313 , 26314 , 26510 , 26511 , 26512		
\dtou		1191
\dump		341
\dviextension		898

- 13639, 13692, 13910, 13921, 13959,
 13962, 13965, 13968, 13999, 14008,
 14017, 14020, 14191, 14204, 14207,
 14214, 14232, 14256, 14257, 14272,
 14282, 14329, 14332, 14341, 14353,
 14364, 14378, 14391, 14431, 14465,
 14485, 14522, 14540, 14543, 14549,
 14563, 14598, 14616, 14619, 14622,
 14625, 14686, 14759, 14829, 14830,
 14839, 14874, 14957, 14961, 14965,
 15027, 15061, 15326, 15355, 15359,
 15519, 15528, 15580, 15591, 15607,
 15615, 15674, 15754, 15765, 15770,
 15804, 15817, 15829, 15835, 16008,
 16015, 16037, 16065, 16080, 16084,
 16106, 16137, 16140, 16165, 16168,
 16205, 16213, 16224, 16227, 16272,
 16287, 16302, 16317, 16332, 16347,
 16368, 16413, 16639, 16677, 16678,
 16687, 16731, 16786, 16787, 16788,
 16892, 16914, 16929, 16947, 16995,
 17011, 17217, 17285, 17290, 17392,
 17413, 17582, 17586, 17686, 17687,
 18141, 18144, 18147, 18157, 18172,
 18199, 18214, 18241, 18257, 18290,
 18298, 18300, 18302, 18304, 18306,
 18308, 18310, 18312, 18330, 18351,
 18355, 18427, 18431, 18627, 18628,
 18633, 18634, 18649, 18656, 18863,
 18873, 18917, 18926, 18938, 18939,
 18941, 18943, 18946, 18947, 18950,
 18951, 18960, 18962, 18964, 18967,
 18968, 18970, 19018, 19021, 19029,
 19037, 19040, 19049, 19052, 19061,
 19069, 19072, 19082, 19188, 19297,
 19341, 19345, 19348, 19359, 19364,
 19463, 19608, 19621, 19709, 19738,
 19777, 19795, 19904, 19938, 20211,
 20229, 20248, 20286, 20342, 20389,
 20393, 20400, 20421, 20432, 20560,
 20682, 20768, 20829, 20903, 20939,
 20951, 20977, 20996, 21192, 21336,
 21361, 21413, 21833, 21835, 21841,
 24020, 24451, 24458, 24469, 25531,
 25543, 25612, 25615, 25618, 25646
 em 195
 \emergencystretch 344
 \enablecjktoken 1228
 \end 118, 287, 345, 14088, 18507, 18508
 end internal commands:
 __regex_end 21180
 \endcsname 14, 21, 39, 43, 49,
 68, 90, 92, 98, 123, 146, 150, 221, 346
 \endgroup 13, 36,
 38, 42, 48, 74, 117, 135, 154, 203, 347
 \endinput 136, 348
 \endL 621
 \endlinechar 220, 233, 349
 \endR 622
 \enquote 14090
 \ensuremath 25366
 \epTeXinputencoding 1192
 \epTeXversion 1193
 \eqno 350
 \errhelp 108, 127, 351
 \errmessage 116, 128, 352
 \ERROR 7393
 \errorcontextlines 353
 \errorstopmode 354
 \escapechar 355
 escapehex 23356
 \ETC 18487
 etex commands:
 \etex_beginL:D 608
 \etex_beginR:D 609
 \etex_botmarks:D 610
 \etex_clubpenalties:D 611
 \etex_currentgrouplevel:D 612
 \etex_currentgroupuypoint:D 613
 \etex_currentifbranch:D 614
 \etex_currentiflevel:D 615
 \etex_currentiftypoint:D 616
 \etex_detokenize:D ... 617, 1427, 1429
 \etex_dimexpr:D 618, 9995, 21752
 \etex_displaywidowpenalties:D .. 620
 \etex_endL:D 621
 \etex_endR:D 622
 \etex_eTeXrevision:D 623
 \etex_eTeXversion:D 624
 \etex_everyeof:D
 625, 3246, 24374, 24401
 \etex_firstmarks:D 626
 \etex_fontchardp:D 627
 \etex_fontcharht:D 628
 \etex_fontcharic:D 629
 \etex_fontcharwd:D 630
 \etex_glueexpr:D 631, 10423,
 10428, 10431, 10440, 10443, 10448,
 10451, 10466, 10472, 10479, 10481,
 10489, 10494, 10499, 10504, 17279
 \etex_glueshrink:D 632, 24249
 \etex_glueshrinkorder:D 633
 \etex_gluestretch:D
 634, 18347, 18353, 24248
 \etex_gluestretchorder:D 635
 \etex_gluetomu:D 636
 \etex_ifcsname:D 637, 1417

- `\etex_ifdefined:D` 638, 803, 1239, 1247,
1278, 1281, 1287, 1346, 1347, 1354,
1366, 1373, 1386, 1394, 1416, 1454
- `\etex_iffontchar:D` 639
- `\etex_interactionmode:D`
..... 640, 21867, 21870, 21872
- `\etex_interlinepenalties:D` 641
- `\etex_lastlinefit:D` 642
- `\etex_lastnodetype:D` 643
- `\etex_marks:D` 644
- `\etex_middle:D` 645
- `\etex_muexpr:D`
..... 646, 10563, 10570, 10573, 10584,
10587, 10592, 10595, 10601, 10606
- `\etex_mutogluue:D`
..... 288, 302, 647, 10563, 10602
- `\etex_numexpr:D` 648, 5262, 11573, 18506
- `\etex_pagediscards:D` 649
- `\etex_parshapedimen:D` 650
- `\etex_parshapeindent:D` 651
- `\etex_parshapelength:D` 652
- `\etex_predisplaydirection:D` ... 653
- `\etex_protected:D`
..... 654, 1461, 1463, 1465,
1466, 1467, 1468, 1469, 1470, 1471,
1472, 1480, 1482, 1484, 1486, 27152
- `\etex_readline:D` 655, 9201
- `\etex_savinghyphcodes:D` 656
- `\etex_savingvdiscards:D` 657
- `\etex_scantokens:D`
..... 658, 3264, 3314, 3329
- `\etex_showgroups:D` 659
- `\etex_showifs:D` 660
- `\etex_showtokens:D`
..... 372, 661, 1390, 3950, 8519
- `\etex_splitbotmarks:D` 662
- `\etex_splitdiscards:D` 663
- `\etex_splitfirstmarks:D` 664
- `\etex_TeXTeXtstate:D` 665
- `\etex_topmarks:D` 666
- `\etex_tracingassigns:D` 667
- `\etex_tracinggroups:D` 668
- `\etex_tracingifs:D` 669
- `\etex_tracingnesting:D` 670
- `\etex_tracingscantokens:D` 671
- `\etex_unexpanded:D`
..... 672, 1383, 1422, 2731, 2735
- `\etex_unless:D` 673, 1407
- `\etex_widowpenalties:D` 674
- `\eTeXrevision` 623
- `\eTeXversion` 624
- `\etoksapp` 901
- `\etokspre` 902
- `\euc` 1194
- `\everycr` 356
- `\everydisplay` 357
- `\everyeof` 625
- `\everyhbox` 358
- `\everyjob` 66, 67, 359
- `\everymath` 360
- `\everypar` 361
- `\everyvbox` 362
- `ex` 195
- `\exhyphenpenalty` 363
- `exp` 191
- exp commands:
- `\exp:w` 32,
32, 33, 306, 312, 327, 328, 329,
335, 364, 365, 370, 382, 443, 449,
518, 622, 624, 625, 628, 629, 647,
652, 652, 985, 1005, 1423, 1843,
1845, 2476, 2489, 2495, 2535, 2539,
2544, 2550, 2556, 2562, 2574, 2580,
2586, 2591, 2593, 2600, 2671, 2676,
2681, 2690, 2692, 2694, 2701, 2709,
2718, 2735, 2739, 2744, 2746, 3521,
3526, 3531, 3536, 3743, 3899, 4136,
4141, 4146, 4151, 4167, 4172, 4177,
4182, 4335, 4344, 4399, 5560, 5565,
5570, 5575, 6316, 6436, 6588, 6596,
6655, 6949, 6957, 7276, 7767, 9036,
9523, 10140, 10175, 10180, 10185,
10190, 11663, 11778, 11782, 11945,
12173, 12299, 12300, 12301, 12302,
12413, 12431, 12459, 12503, 12515,
12520, 12528, 12537, 12559, 12565,
12637, 12650, 12651, 12660, 12673,
12691, 12692, 12712, 12725, 12729,
12751, 12779, 12792, 12805, 12829,
12840, 12850, 12869, 12882, 12895,
12898, 12910, 12933, 12962, 12976,
12992, 13012, 13023, 13029, 13039,
13085, 13092, 13123, 13138, 13146,
13163, 13179, 13183, 13192, 13229,
13238, 13247, 13252, 13254, 13265,
13267, 13282, 13285, 13292, 13303,
13390, 13410, 13428, 13431, 13445,
13458, 13506, 13524, 13595, 13607,
13636, 13638, 13642, 13644, 13702,
13712, 13722, 13734, 13741, 13761,
13901, 13918, 13928, 14083, 14084,
14085, 14276, 14279, 14287, 14297,
14305, 15299, 15828, 15850, 16010,
16183, 16389, 17052, 17067, 17084,
17121, 17138, 17180, 17199, 17212,
17244, 17259, 17270, 17337, 17460,
17468, 17491, 17507, 24318, 24325,

- 24489, 24530, 24544, 25400, 25406,
25412, 25418, 25434, 25454, 25536
\exp_after:wN 30, 31, 32, 306,
326, 328, 370, 372, 426, 505, 599,
621, 622, 624, 625, 690, 691, 750,
802, 824, 891, 1005, 1420, 1438,
1440, 1445, 1447, 1653, 1683, 1684,
1696, 1745, 1748, 1752, 1824, 1830,
1843, 1845, 1897, 1910, 1928, 1930,
1980, 1985, 1992, 2029, 2033, 2038,
2049, 2065, 2067, 2070, 2093, 2095,
2098, 2224, 2226, 2235, 2255, 2257,
2296, 2360, 2453, 2463, 2470, 2472,
2475, 2476, 2488, 2489, 2494, 2495,
2500, 2505, 2507, 2510, 2519, 2521,
2524, 2525, 2526, 2529, 2531, 2533,
2535, 2538, 2543, 2548, 2549, 2550,
2554, 2555, 2556, 2560, 2561, 2562,
2566, 2567, 2568, 2572, 2573, 2574,
2578, 2579, 2580, 2584, 2585, 2586,
2590, 2591, 2592, 2593, 2597, 2598,
2599, 2600, 2604, 2605, 2606, 2611,
2612, 2613, 2614, 2618, 2619, 2620,
2621, 2667, 2670, 2671, 2675, 2676,
2680, 2681, 2688, 2690, 2692, 2694,
2696, 2699, 2700, 2704, 2707, 2708,
2712, 2715, 2716, 2717, 2725, 2728,
2730, 2732, 2733, 2735, 2738, 2743,
2775, 2779, 2801, 2808, 2828, 2979,
2997, 3008, 3262, 3263, 3312, 3313,
3327, 3328, 3387, 3388, 3389, 3394,
3430, 3441, 3442, 3510, 3512, 3601,
3714, 3741, 3772, 3777, 3778, 3779,
3781, 3794, 3804, 3821, 3843, 3853,
3856, 3873, 3874, 3884, 3889, 3890,
3905, 3906, 3907, 3950, 3951, 4197,
4198, 4210, 4265, 4288, 4322, 4323,
4334, 4335, 4343, 4351, 4353, 4360,
4365, 4383, 4384, 4385, 4397, 4398,
4425, 4427, 4433, 4439, 4453, 4473,
4484, 4500, 4508, 4516, 4523, 4530,
4542, 4613, 4619, 4621, 4645, 4688,
4689, 4815, 4817, 4829, 4837, 4934,
4968, 4980, 4993, 4994, 4995, 5017,
5018, 5063, 5092, 5093, 5094, 5151,
5152, 5178, 5179, 5182, 5275, 5280,
5288, 5289, 5301, 5302, 5323, 5324,
5350, 5351, 5493, 5498, 5503, 5526,
5528, 5681, 5682, 5683, 5708, 5709,
5892, 5920, 5925, 5953, 5966, 5976,
6003, 6005, 6006, 6014, 6031, 6075,
6163, 6172, 6224, 6226, 6233, 6314,
6331, 6336, 6340, 6437, 6511, 6587,
6595, 6655, 6727, 6742, 6764, 6778,
6839, 6847, 6853, 6948, 6956, 7040,
7041, 7044, 7045, 7276, 7277, 7324,
7332, 7360, 7361, 7404, 7405, 7406,
7517, 7536, 7583, 7621, 7650, 7651,
7653, 7659, 7662, 7724, 7726, 7736,
7737, 7738, 7740, 7746, 7747, 7749,
7756, 7757, 7759, 7765, 7766, 7769,
7846, 7855, 7864, 7957, 7968, 8139,
8140, 8141, 8332, 8519, 8520, 9037,
9038, 9039, 9040, 9052, 9128, 9280,
9453, 9456, 9506, 9515, 9518, 9521,
9522, 9524, 9563, 9620, 9649, 9658,
9679, 9689, 9694, 9833, 9846, 10079,
10083, 10091, 10092, 10104, 10105,
10135, 10140, 10151, 10154, 10276,
10277, 10278, 10340, 10471, 10638,
10639, 10648, 10650, 10667, 10672,
10674, 10691, 10700, 10704, 10705,
10712, 10714, 10715, 10716, 10717,
10728, 10751, 10754, 10832, 10875,
11025, 11026, 11334, 11414, 11415,
11434, 11445, 11458, 11459, 11623,
11625, 11626, 11637, 11638, 11639,
11649, 11651, 11659, 11661, 11668,
11669, 11670, 11671, 11672, 11673,
11678, 11679, 11680, 11681, 11682,
11683, 11684, 11727, 11740, 11743,
11754, 11756, 11771, 11775, 11776,
11777, 11780, 11781, 11845, 11847,
11874, 11878, 11903, 11907, 11924,
11931, 11933, 11945, 12028, 12036,
12053, 12062, 12108, 12173, 12242,
12243, 12244, 12304, 12314, 12333,
12339, 12365, 12366, 12369, 12380,
12384, 12391, 12392, 12403, 12404,
12413, 12420, 12422, 12423, 12431,
12459, 12477, 12478, 12481, 12482,
12484, 12485, 12489, 12490, 12492,
12493, 12502, 12503, 12508, 12514,
12520, 12528, 12537, 12557, 12558,
12561, 12562, 12564, 12571, 12572,
12574, 12592, 12593, 12621, 12624,
12629, 12630, 12635, 12636, 12638,
12647, 12648, 12649, 12650, 12653,
12654, 12655, 12658, 12673, 12690,
12691, 12701, 12702, 12712, 12724,
12728, 12741, 12743, 12751, 12761,
12763, 12769, 12774, 12776, 12778,
12784, 12785, 12789, 12791, 12803,
12804, 12826, 12828, 12834, 12837,
12839, 12843, 12848, 12853, 12854,
12864, 12865, 12867, 12868, 12871,
12875, 12880, 12894, 12897, 12909,
12918, 12925, 12926, 12927, 12928,

12930, 12932, 12943, 12944, 12945,
12946, 12948, 12950, 12952, 12953,
12954, 12960, 12961, 12971, 12975,
12976, 12978, 12979, 12980, 12985,
12991, 13002, 13004, 13011, 13012,
13014, 13015, 13022, 13028, 13038,
13106, 13119, 13120, 13121, 13122,
13136, 13137, 13139, 13144, 13145,
13160, 13162, 13179, 13183, 13192,
13226, 13227, 13228, 13234, 13235,
13236, 13237, 13243, 13244, 13245,
13246, 13253, 13266, 13274, 13280,
13281, 13283, 13284, 13290, 13291,
13293, 13319, 13332, 13354, 13355,
13357, 13358, 13365, 13366, 13368,
13371, 13383, 13385, 13386, 13388,
13389, 13391, 13403, 13404, 13405,
13408, 13409, 13410, 13420, 13421,
13422, 13425, 13426, 13427, 13430,
13434, 13443, 13444, 13455, 13456,
13457, 13460, 13462, 13463, 13464,
13489, 13490, 13492, 13493, 13494,
13504, 13505, 13506, 13508, 13509,
13510, 13522, 13523, 13526, 13528,
13529, 13530, 13550, 13551, 13552,
13553, 13554, 13555, 13556, 13566,
13567, 13569, 13570, 13571, 13577,
13588, 13589, 13590, 13591, 13592,
13593, 13594, 13595, 13600, 13601,
13602, 13603, 13604, 13605, 13606,
13622, 13623, 13625, 13626, 13633,
13634, 13635, 13640, 13641, 13643,
13659, 13674, 13683, 13693, 13699,
13700, 13701, 13706, 13719, 13720,
13721, 13727, 13739, 13740, 13755,
13757, 13758, 13759, 13900, 13917,
13927, 13952, 13953, 14000, 14082,
14190, 14192, 14231, 14233, 14236,
14271, 14273, 14275, 14278, 14285,
14286, 14289, 14290, 14295, 14296,
14303, 14304, 14337, 14338, 14339,
14341, 14352, 14377, 14379, 14385,
14386, 14390, 14393, 14415, 14417,
14430, 14432, 14438, 14440, 14443,
14449, 14451, 14453, 14454, 14455,
14457, 14462, 14464, 14466, 14470,
14473, 14479, 14480, 14484, 14486,
14487, 14488, 14496, 14498, 14499,
14506, 14512, 14519, 14520, 14525,
14526, 14527, 14528, 14547, 14548,
14549, 14555, 14556, 14557, 14562,
14564, 14572, 14574, 14576, 14577,
14579, 14590, 14592, 14594, 14595,
14600, 14651, 14652, 14659, 14660,
14662, 14664, 14666, 14669, 14672,
14674, 14676, 14685, 14687, 14693,
14695, 14697, 14698, 14699, 14705,
14707, 14709, 14710, 14711, 14732,
14733, 14736, 14744, 14746, 14750,
14751, 14752, 14753, 14758, 14760,
14767, 14770, 14773, 14776, 14785,
14788, 14791, 14794, 14801, 14803,
14809, 14817, 14819, 14821, 14838,
14840, 14847, 14849, 14852, 14858,
14860, 14862, 14863, 14864, 14866,
14880, 14881, 14884, 14902, 14904,
14906, 14918, 14921, 14924, 14927,
14930, 14933, 14936, 14939, 14943,
14955, 14959, 14963, 14966, 14981,
14987, 14989, 14991, 15001, 15025,
15028, 15040, 15042, 15046, 15047,
15048, 15050, 15051, 15053, 15059,
15060, 15066, 15069, 15070, 15071,
15072, 15080, 15122, 15127, 15129,
15136, 15139, 15142, 15145, 15148,
15151, 15159, 15160, 15172, 15180,
15182, 15192, 15194, 15201, 15210,
15212, 15215, 15218, 15221, 15224,
15237, 15239, 15247, 15249, 15257,
15259, 15269, 15272, 15275, 15282,
15297, 15298, 15315, 15317, 15318,
15375, 15388, 15390, 15396, 15409,
15411, 15413, 15437, 15451, 15453,
15460, 15462, 15503, 15504, 15505,
15507, 15508, 15509, 15511, 15512,
15518, 15520, 15521, 15527, 15529,
15530, 15531, 15532, 15544, 15550,
15552, 15587, 15594, 15601, 15621,
15622, 15624, 15626, 15628, 15641,
15646, 15647, 15648, 15649, 15650,
15654, 15659, 15661, 15667, 15673,
15675, 15676, 15682, 15683, 15684,
15685, 15686, 15687, 15688, 15689,
15694, 15696, 15698, 15700, 15702,
15707, 15709, 15711, 15713, 15715,
15717, 15735, 15739, 15747, 15748,
15753, 15755, 15764, 15767, 15768,
15769, 15771, 15772, 15773, 15781,
15787, 15799, 15802, 15803, 15805,
15806, 15830, 15831, 15834, 15836,
15852, 15856, 15857, 15858, 15874,
15880, 15889, 15890, 15891, 15899,
15915, 15931, 15947, 15963, 15979,
16005, 16009, 16010, 16014, 16016,
16052, 16058, 16059, 16061, 16063,
16064, 16066, 16067, 16077, 16078,
16081, 16082, 16083, 16085, 16086,
16087, 16104, 16105, 16107, 16108,

- 16114, 16116, 16119, 16122, 16125,
 16128, 16136, 16139, 16141, 16144,
 16151, 16155, 16163, 16164, 16167,
 16169, 16171, 16175, 16179, 16184,
 16185, 16193, 16194, 16195, 16196,
 16361, 16362, 16363, 16365, 16383,
 16384, 16385, 16386, 16387, 16388,
 16395, 16404, 16411, 16412, 16422,
 16586, 16587, 16588, 16609, 16610,
 16620, 16629, 16637, 16638, 16640,
 16641, 16646, 16647, 16656, 16663,
 16672, 16673, 16686, 16688, 16716,
 16717, 16726, 16729, 16754, 16760,
 16761, 16803, 16804, 16806, 16820,
 16821, 16829, 16840, 16874, 16877,
 16887, 16888, 16891, 16893, 16899,
 16913, 16915, 16956, 16959, 16979,
 17052, 17062, 17066, 17084, 17087,
 17108, 17109, 17116, 17120, 17138,
 17141, 17170, 17171, 17177, 17178,
 17179, 17186, 17194, 17198, 17212,
 17215, 17231, 17232, 17239, 17243,
 17254, 17258, 17270, 17276, 17277,
 17278, 17284, 17286, 17289, 17291,
 17333, 17335, 17336, 17357, 17382,
 17388, 17391, 17394, 17397, 17404,
 17410, 17414, 17424, 17435, 17455,
 17462, 17466, 17467, 17478, 17485,
 17494, 17495, 17503, 17511, 17512,
 17600, 17601, 17602, 17610, 17628,
 17629, 17677, 17775, 17776, 17818,
 17837, 17838, 17853, 17854, 17855,
 18075, 18081, 18083, 18094, 18154,
 18155, 18156, 18157, 18163, 18164,
 18180, 18198, 18200, 18206, 18207,
 18238, 18240, 18242, 18272, 18287,
 18289, 18291, 18316, 18326, 18334,
 18344, 18354, 18356, 18358, 18393,
 18417, 18426, 18429, 18430, 18432,
 18433, 18441, 18442, 18456, 18520,
 18533, 18534, 18539, 18540, 18543,
 18546, 18584, 18591, 18598, 18604,
 18611, 18619, 18655, 18657, 18666,
 18798, 18801, 18842, 18862, 18864,
 18865, 18872, 18875, 18876, 18900,
 18919, 18928, 19028, 19030, 19036,
 19039, 19041, 19048, 19051, 19053,
 19060, 19062, 19068, 19071, 19074,
 19391, 19464, 19476, 19607, 19610,
 19620, 19622, 19804, 19812, 19887,
 19937, 20162, 20462, 20590, 20613,
 20670, 20696, 20760, 20761, 20769,
 20772, 20937, 20938, 20941, 20942,
 20950, 20952, 20953, 20962, 20976,
 20979, 21054, 21303, 21335, 21337,
 23589, 23955, 23980, 24048, 24049,
 24111, 24121, 24126, 24129, 24323,
 24343, 24377, 24378, 24487, 24528,
 24542, 24559, 24560, 24724, 24725,
 24746, 24747, 24795, 24798, 24799,
 24846, 24881, 24973, 25023, 25101,
 25103, 25109, 25131, 25144, 25146,
 25157, 25159, 25281, 25339, 25400,
 25406, 25412, 25418, 25432, 25435,
 25452, 25454, 25455, 25462, 25467,
 25469, 25472, 25486, 25532, 25534,
 25535, 25536, 25541, 25542, 25544,
 25554, 25555, 25562, 25563, 25640,
 25645, 25647, 25653, 25715, 25719
 \exp_args:cc
 26, 1437, 1933, 1944, 1950, 1956, 2518
 \exp_args:Nc 24, 26, 322, 1437, 1441,
 1449, 1591, 1629, 1638, 1657, 1678,
 1688, 1709, 1714, 2159, 2178, 2204,
 2209, 2216, 2275, 2287, 2359, 2392,
 2393, 2394, 2395, 2417, 2421, 2518,
 2772, 3041, 3356, 3571, 3971, 6166,
 6185, 9216, 11520, 12322, 12553,
 13414, 13438, 13468, 13470, 13472,
 13474, 13476, 13478, 13480, 13498,
 13514, 13515, 13534, 13536, 18376,
 19759, 25384, 25476, 25481, 25489
 \exp_args:Ncc 27, 2206, 2210,
 2218, 2400, 2401, 2402, 2403, 2518
 \exp_args:Nccc 28, 2518
 \exp_args:Ncco 28, 2595
 \exp_args:Nccx 29, 2641
 \exp_args:Ncf 27, 2546
 \exp_args:NcNc 28, 2595
 \exp_args:NcNo 28, 2595
 \exp_args:Ncno 28, 2641
 \exp_args:NcnV 28, 2641
 \exp_args:Ncnx 29, 2641
 \exp_args:Nco 27, 330, 2546
 \exp_args:Ncoo 28, 2641
 \exp_args:NcV 27, 2546
 \exp_args:Ncv 27, 2546
 \exp_args:NcVV 28, 2641
 \exp_args:Ncx 28, 2624, 9045
 \exp_args:Nf 27,
 2263, 2534, 3911, 3912, 3928, 4312,
 4314, 4373, 4375, 4391, 5066, 5067,
 5083, 5561, 5566, 5571, 5576, 5766,
 5835, 5837, 5855, 5864, 5875, 5884,
 6022, 6039, 7029, 7080, 7094, 7115,
 7158, 7228, 7234, 7240, 7246, 9050,
 9555, 10176, 10181, 10186, 10191,
 14143, 17048, 17531, 17540, 17545,

- 20317, 23693, 24027, 24029, 24053,
 24055, 24342, 24427, 24441, 24930,
 24938, 24993, 25513, 25521, 25525,
 25569, 25575, 25585, 26067, 26595
 \exp_args:Nff . 28, [2624](#), 11971, 25519
 \exp_args:Nffo 28, [2641](#)
 \exp_args:Nfo 28, [2624](#)
 \exp_args:NNc . 27, 295, 2205, 2208,
 2217, 2289, 2396, 2397, 2398, 2399,
 2434, 2437, [2518](#), 5723, 5734, 9052,
 9128, 9280, 10307, 10314, 14153, 14160
 \exp_args:Nnc 28, [2624](#)
 \exp_args:NNcf 28, [2641](#)
 \exp_args:NNf 27,
 1649, [2546](#), 9127, 9279, 9441, 10300,
 16355, 16356, 25892, 26213, 26351
 \exp_args:Nnf 28, [2624](#), 11518
 \exp_args:Nnff 28, [2641](#)
 \exp_args:Nnnc 28, [2641](#)
 \exp_args:Nnnf 28, [2641](#)
 \exp_args:NNNo
 . . . 28, [2529](#), 3253, 19123, 20011,
 20068, 21213, 21297, 24380, 24405
 \exp_args:NNno 28, [2641](#)
 \exp_args:Nnno 28, [2641](#)
 \exp_args:NNNV 28, [2595](#), 9721
 \exp_args:NNnV 28, [2641](#)
 \exp_args:NNNx
 29, [786](#), [2641](#), 19131, 19603
 \exp_args:NNnx 29, [2641](#)
 \exp_args:Nnnx 29, [2641](#)
 \exp_args:NNo
 22, 27, [2529](#), 5754, 7962, 20857
 \exp_args:Nno
 . 28, [2624](#), 3267, 6984, 7838, 9198,
 10143, 12013, 12021, 12030, 12047,
 12055, 12083, 12580, 12584, 24384
 \exp_args:NNoo 28, [2641](#)
 \exp_args:NNox 29, [2641](#)
 \exp_args:Nnox 29, [2641](#)
 \exp_args:NNV 27, [2546](#)
 \exp_args:NNv 27, [2546](#)
 \exp_args:NnV 28, [2624](#)
 \exp_args:Nnv 28, [2624](#)
 \exp_args:NNVV 28, [2641](#)
 \exp_args:NNx 28, 2442, [2624](#), 9891, 9905
 \exp_args:Nnx 28, [2624](#), 9248
 \exp_args:No
 . . 24, 26, 1005, 1539, 1547, 2426,
 2431, [2529](#), 3030, 3246, 3252, 3475,
 3476, 3477, 3503, 3504, 3505, 3506,
 3507, 3559, 3578, 3587, 3645, 3648,
 3650, 3760, 3762, 3788, 3797, 3930,
 4204, 4231, 4236, 4250, 4308, 4319,
 4369, 4380, 4447, 4466, 4504, 4519,
 4807, 5754, 5841, 5847, 6846, 6858,
 6860, 6895, 6900, 7109, 7113, 8513,
 9231, 9342, 9372, 9448, 9855, 10477,
 11122, 11140, 11168, 11190, 18380,
 18403, 18460, 18462, 19199, 19256,
 19277, 19912, 20291, 20904, 20948,
 21358, 21388, 21867, 24374, 25496,
 25498, 25501, 25505, 25583, 25592
 \exp_args:Noc 28, [2624](#)
 \exp_args:Nof 28, [2624](#)
 \exp_args:Noo . 28, [2624](#), 19300, 20304
 \exp_args:Noof 28, [2641](#)
 \exp_args:Nooo 28, [2641](#)
 \exp_args:Noox 29, [2641](#)
 \exp_args:Nox 28, [2624](#)
 \exp_args:NV
 26, [2534](#), 9589, 9756, 11120,
 11138, 11166, 11188, 18693, 25710
 \exp_args:Nv 27, [2534](#)
 \exp_args:NVo 28, [2624](#)
 \exp_args:NVV 27, [2546](#), 9501
 \exp_args:Nx 27, 2233, [2623](#),
 3939, 6187, 6293, 7383, 8294, 9378,
 11124, 11142, 11170, 11192, 13875,
 19433, 19434, 19815, 20707, 25816
 \exp_args:Nxo 28, [2624](#)
 \exp_args:Nxx 28, [2624](#)
 \exp_args_generate:n 237, [3028](#)
 \exp_end: . . . 32, 32, 306, 309, 312,
 328, 329, 335, 364, 365, 370, 382,
 445, 518, 622, 652, 980, 1004, 1005,
 1424, 1934, 1945, 1951, 1957, 2507,
 2516, [2746](#), 3551, 3721, 3905, 3906,
 4194, 4368, 5587, 6468, 6471, 6472,
 6473, 6474, 6475, 6476, 6477, 6478,
 6479, 6481, 6583, 7307, 7321, 7324,
 7329, 7332, 7336, 7344, 7401, 7405,
 9040, 10202, 12304, 13259, 15852,
 17462, 24517, 25390, 25423, 25544
 \exp_end_continue_f:nw 33, [2746](#)
 \exp_end_continue_f:w
 32, 33, 327, 624, 625, 2476,
 2535, 2562, 2586, 2681, 2694, 2735,
[2746](#), 6316, 6655, 7767, 8984, 9523,
 10140, 11663, 11778, 11782, 11945,
 12413, 12431, 12451, 12515, 12520,
 12528, 12537, 12559, 12637, 12673,
 12681, 12712, 13085, 13092, 13138,
 13185, 13192, 13229, 13273, 13279,
 13282, 13292, 13303, 13445, 13636,
 13638, 13642, 13644, 13702, 13712,
 13722, 13734, 13901, 13918, 13928,
 14083, 14084, 14085, 14276, 14287,

- 14297, 14305, 15299, 16010, 16183,
 16389, 17052, 17067, 17084, 17121,
 17138, 17180, 17199, 17212, 17244,
 17259, 17270, 17338, 17468, 24318
 $\backslash\exp_last_two_unbraced:Nnn$
 29, 2727, 22710, 22949, 22953
 $\backslash\exp_last_unbraced:Nn$
 29, 29, 2688, 3324, 5853,
 5873, 7408, 11966, 14367, 18853,
 23078, 23083, 23161, 23167, 25768
 $\backslash\exp_last_unbraced:NNn$
 29, 2688, 3728,
 6934, 6967, 9375, 22920, 24550, 24609
 $\backslash\exp_last_unbraced:Nnn$
 29, 2688, 8020, 8104, 24095
 $\backslash\exp_last_unbraced:NNNn$ 29, 2688
 $\backslash\exp_last_unbraced:NnNn$ 29, 2688
 $\backslash\exp_last_unbraced:NNNNn$
 29, 2688, 2788, 2792,
 2970, 4562, 9650, 10886, 11731, 11749
 $\backslash\exp_not:N$ 30, 30,
 147, 196, 328, 335, 349, 351, 368,
 369, 476, 483, 630, 802, 811, 883,
 887, 981, 1420, 1772, 1776, 1901,
 1967, 1970, 2295, 2296, 2359, 2360,
 2463, 2500, 2732, 2732, 2765, 2767,
 2768, 2775, 2776, 2777, 2778, 2779,
 2780, 2781, 2782, 2784, 2785, 2786,
 2812, 2821, 2873, 2874, 2875, 2876,
 2950, 2956, 2957, 2958, 2960, 2963,
 2966, 2967, 2979, 3052, 3247, 3309,
 3380, 3801, 3804, 3818, 3821, 3852,
 3859, 3999, 4000, 4222, 4223, 4806,
 4808, 5139, 5739, 6002, 6909, 7019,
 7022, 7030, 7031, 7264, 7348, 7352,
 7386, 7436, 7440, 7445, 7450, 7455,
 7462, 7469, 7474, 7479, 7484, 7489,
 7494, 7504, 7509, 7514, 7517, 7518,
 7519, 7521, 7522, 7531, 7536, 7551,
 7552, 7569, 7574, 7575, 7576, 7577,
 7578, 7579, 7581, 7583, 7584, 7585,
 7586, 7589, 7590, 7593, 7594, 7615,
 7618, 7619, 7621, 7622, 7623, 7626,
 7627, 7629, 7630, 7632, 7633, 7635,
 7698, 7704, 7735, 7738, 7745, 7746,
 7755, 7756, 7785, 7951, 8059, 8082,
 8401, 8403, 8405, 8407, 8412, 8413,
 8418, 8420, 8422, 8682, 8684, 8686,
 8688, 8693, 8694, 8699, 8701, 8703,
 10319, 10345, 10724, 10733, 10894,
 10896, 10910, 10912, 10969, 10970,
 11093, 11094, 11727, 11728, 11729,
 12476, 12477, 12574, 12575, 12576,
 12577, 12578, 12683, 12723, 12727,
 12749, 12842, 12874, 12959, 12973,
 12990, 13000, 13010, 13047, 13050,
 13156, 13157, 13159, 13160, 13161,
 13162, 13163, 13164, 13167, 13169,
 13171, 13350, 13352, 13380, 13381,
 13383, 13384, 13385, 13386, 13387,
 13388, 13389, 13390, 13391, 13392,
 13393, 13488, 13503, 14165, 14320,
 16242, 16243, 16244, 16248, 16249,
 16250, 17332, 17333, 17335, 17336,
 17337, 17338, 17339, 17341, 18142,
 18145, 18297, 18298, 18299, 18300,
 18301, 18302, 18303, 18304, 18305,
 18306, 18307, 18308, 18309, 18310,
 18311, 18312, 18315, 18316, 18317,
 18829, 18831, 18833, 18835, 18837,
 18839, 19187, 19189, 19384, 19386,
 19397, 19401, 19567, 20020, 20701,
 20918, 21046, 21059, 21422, 23991,
 24058, 24060, 24064, 24066, 24071,
 24073, 24163, 24173, 24375, 24401,
 25102, 25104, 25145, 25147, 25158,
 25160, 25636, 25637, 25838, 25839,
 25847, 26174, 26194, 27154, 27157
 $\backslash\exp_not:n$ 15,
 26, 27, 30, 30, 30, 30, 31, 31, 31,
 43, 44, 44, 46, 67, 71, 72, 110,
 110, 112, 130, 147, 236, 240, 241,
 244, 246, 248, 248, 249, 289, 303,
 347, 349, 354, 363, 399, 402, 450,
 451, 454, 457, 492, 797, 801, 802,
 807, 811, 819, 824, 883, 889, 889,
 893, 1005, 1006, 1008, 1420, 1775,
 1902, 1904, 1972, 2235, 2463, 2483,
 2685, 2732, 2880, 2895, 2910, 3066,
 3116, 3124, 3142, 3145, 3151, 3154,
 3166, 3169, 3172, 3175, 3178, 3181,
 3184, 3187, 3198, 3201, 3204, 3207,
 3210, 3213, 3216, 3219, 3269, 3311,
 3318, 3377, 3381, 3382, 3383, 3640,
 3642, 3927, 3939, 4037, 4101, 4243,
 4828, 4829, 4836, 4837, 4853, 4885,
 4905, 4908, 4911, 4973, 5005, 5029,
 5082, 5140, 5194, 5204, 5740, 6622,
 6664, 6665, 6679, 6681, 6751, 6846,
 6854, 6874, 6910, 7023, 7029, 7057,
 7062, 7093, 7124, 7387, 7588, 7699,
 7705, 7784, 7786, 7952, 8027, 8060,
 8063, 8064, 8083, 8087, 8395, 8412,
 8545, 8676, 8693, 9336, 9353, 9735,
 10320, 10663, 10665, 10687, 10689,
 10724, 10734, 10735, 10767, 10932,
 10972, 10996, 11095, 11240, 11265,
 11467, 11469, 12569, 13836, 13838,

- 13840, 14166, 14321, 16422, 17916,
18170, 18284, 18314, 19187, 19189,
19818, 20077, 20221, 20446, 20690,
20702, 20757, 20976, 20979, 20988,
21046, 21054, 21071, 21086, 21127,
21310, 21315, 23292, 23295, 23297,
23586, 24120, 25390, 25391, 25400,
25406, 25412, 25418, 25432, 25452,
25465, 25472, 25699, 26174, 26194
- `\exp_stop_f`: 31, 32, 88, 327,
366, 399, 591, 603, 670, 671, 797,
802, 2473, 4102, 4109, 4116, 4350,
4366, 4405, 4406, 4412, 4424, 4440,
4441, 4840, 5277, 5292, 5305, 5520,
5525, 5704, 7154, 7156, 7224, 7226,
7230, 7232, 7236, 7238, 7242, 7244,
7282, 7283, 7290, 7291, 7292, 7293,
7298, 7299, 7318, 7333, 7339, 7408,
7957, 9052, 9128, 9280, 9511, 9525,
10149, 11576, 11771, 11886, 11901,
11926, 12173, 12177, 12181, 12183,
12187, 12191, 12199, 12204, 12217,
12224, 12237, 12248, 12249, 12260,
12261, 12270, 12273, 12284, 12326,
12382, 12387, 12458, 12488, 12646,
12689, 12740, 12760, 12787, 12801,
12836, 12863, 12872, 12891, 12907,
12923, 12941, 13001, 13020, 13036,
13051, 13065, 13274, 13353, 13364,
13617, 13621, 13966, 13972, 13974,
13989, 13998, 14006, 14014, 14015,
14212, 14330, 14336, 14351, 14388,
14461, 14483, 14537, 14538, 14546,
14883, 14901, 14954, 14958, 14962,
14980, 15015, 15016, 15017, 15018,
15019, 15045, 15057, 15074, 15352,
15353, 15450, 15543, 15576, 15589,
15594, 15603, 15605, 15732, 15761,
15766, 15796, 15833, 15873, 15892,
16004, 16050, 16051, 16056, 16062,
16080, 16103, 16135, 16138, 16181,
16201, 16207, 16222, 16263, 16278,
16293, 16308, 16323, 16338, 16366,
16410, 16636, 16788, 16790, 16827,
16839, 16871, 16912, 16921, 16936,
16955, 16988, 17001, 17085, 17139,
17187, 17190, 17213, 17443, 17447,
17454, 17579, 17583, 17676, 17935,
18108, 18117, 18118, 18152, 18222,
18230, 18253, 18255, 18256, 18260,
18277, 18328, 18331, 18347, 18353,
18425, 18428, 18626, 18627, 18628,
18634, 18654, 18915, 18935, 18936,
18940, 18944, 18945, 18948, 18949,
18957, 18958, 18961, 18965, 18966,
18969, 19016, 19111, 19357, 19362,
19376, 19377, 19390, 19461, 19462,
19501, 19600, 19776, 19935, 20226,
20244, 20273, 20283, 20338, 20351,
20362, 20378, 20429, 20633, 20767,
20771, 20838, 20901, 20914, 20935,
20940, 20946, 20993, 21050, 21067,
21090, 21308, 21313, 21334, 21412,
21424, 21429, 23507, 23508, 23538,
23539, 23733, 24264, 24446, 24447,
24452, 24931, 24946, 25001, 25531,
25540, 25553, 25561, 25609, 25610,
25616, 25636, 25637, 25638, 25639
- exp internal commands:
- `__exp_arg_last_unbraced:nn` . . . 2665
`__exp_arg_next:Nnn` 2463, 2470
`__exp_arg_next:nnn`
. . . 328, 2463, 2472, 2475, 2488, 2494
`__exp_eval_error_msg:w` 2498
`__exp_eval_register:N`
. 2489, 2495, 2498,
2539, 2544, 2550, 2556, 2574, 2580,
2592, 2593, 2600, 2671, 2676, 2690,
2692, 2701, 2709, 2718, 2739, 2744
`\l__exp_internal_tl`
. 296, 1488, 1492, 1493,
2463, 2463, 2482, 2484, 2685, 2686
`__exp_last_two_unbraced:nnN` . . 2727
`\expandafter` 13,
14, 21, 38, 39, 42, 43, 48, 49, 66, 67,
90, 92, 98, 123, 146, 154, 169, 185, 364
`\expanded` 905
`\expandglyphsinfont` 993
`\ExplFileDate` 6
`\ExplFileDescription` 6
`\ExplFileName` 6
`\ExplFileVersion` 6
`\explicitdiscretionary` 906
`\explicitthyphenpenalty` 903
`\ExplSyntaxOff` 3, 6, 6,
104, 206, 239, 253, 263, 263, 264, 289
`\ExplSyntaxOn` 3, 6, 6,
104, 235, 263, 263, 264, 289, 351, 468
- F**
- false 196
`\fam` 365
`\fi` 17, 35, 41,
51, 70, 71, 72, 95, 96, 97, 100, 101,
130, 139, 152, 153, 170, 186, 204, 366
- fi commands:
- `\fi`: 20, 88, 89, 89, 93, 99, 99,
148, 164, 224, 224, 225, 306, 312,

313, 355, 356, 382, 384, 387, 416,
426, 445, 603, 629, 644, 676, 737,
802, 1402, 1448, 1533, 1643, 1660,
1671, 1672, 1720, 1753, 1754, 1898,
1911, 1931, 1981, 1986, 1993, 2028,
2033, 2059, 2060, 2068, 2074, 2087,
2088, 2096, 2102, 2227, 2248, 2258,
2272, 2329, 2390, 2454, 2503, 2506,
2513, 2514, 2753, 2786, 2802, 2809,
2818, 2832, 2833, 2838, 2839, 2840,
2857, 2858, 2859, 2860, 2861, 2862,
2863, 2864, 2872, 2892, 2894, 2926,
2927, 2928, 2996, 3007, 3017, 3393,
3394, 3424, 3435, 3446, 3456, 3471,
3482, 3486, 3516, 3715, 3768, 3773,
3782, 3784, 3811, 3832, 3848, 3857,
3866, 3872, 3879, 3884, 3895, 3899,
3907, 4103, 4110, 4116, 4211, 4279,
4283, 4284, 4302, 4355, 4368, 4410,
4416, 4417, 4428, 4441, 4442, 4463,
4501, 4527, 4614, 4622, 4646, 4660,
4668, 4693, 4881, 4884, 4921, 4935,
4952, 4962, 5016, 5021, 5280, 5313,
5335, 5336, 5338, 5393, 5402, 5478,
5486, 5513, 5521, 5527, 5556, 5596,
5606, 5706, 5921, 5954, 6002, 6007,
6144, 6176, 6217, 6227, 6277, 6309,
6324, 6346, 6483, 6485, 6487, 6489,
6491, 6493, 6728, 6743, 6766, 6780,
7287, 7290, 7291, 7292, 7293, 7298,
7299, 7304, 7305, 7306, 7343, 7352,
7410, 7412, 7441, 7446, 7451, 7456,
7463, 7470, 7475, 7480, 7485, 7490,
7495, 7500, 7505, 7510, 7532, 7543,
7544, 7593, 7594, 7645, 7654, 7663,
7671, 7727, 7741, 7750, 7760, 7770,
8121, 8939, 8943, 8944, 9188, 9463,
9504, 9513, 9534, 9543, 9547, 9554,
9562, 10083, 10116, 10131, 10148,
10152, 10166, 10169, 10668, 10692,
10706, 10718, 10743, 10744, 10755,
11627, 11636, 11640, 11652, 11662,
11757, 11810, 11813, 11814, 11819,
11833, 11871, 11872, 11873, 11874,
11875, 11876, 11877, 11878, 11879,
11880, 11881, 11882, 11895, 11897,
11908, 11911, 11925, 11930, 11934,
12088, 12179, 12180, 12189, 12190,
12201, 12202, 12203, 12214, 12215,
12216, 12223, 12234, 12235, 12236,
12246, 12247, 12251, 12252, 12260,
12263, 12264, 12272, 12283, 12303,
12326, 12370, 12389, 12390, 12399,
12405, 12425, 12426, 12453, 12462,
12479, 12486, 12494, 12495, 12598,
12599, 12600, 12603, 12606, 12645,
12661, 12687, 12688, 12695, 12703,
12732, 12733, 12736, 12738, 12739,
12744, 12754, 12757, 12759, 12764,
12795, 12808, 12813, 12819, 12822,
12823, 12857, 12858, 12885, 12886,
12899, 12902, 12913, 12936, 12955,
12965, 12981, 12995, 13001, 13005,
13010, 13016, 13031, 13042, 13061,
13071, 13073, 13079, 13100, 13128,
13151, 13184, 13186, 13309, 13359,
13363, 13373, 13374, 13393, 13411,
13432, 13435, 13465, 13495, 13511,
13531, 13572, 13584, 13597, 13599,
13619, 13620, 13627, 13645, 13684,
13694, 13912, 13923, 13952, 13953,
13954, 13961, 13963, 13964, 13970,
13971, 13974, 14001, 14009, 14010,
14018, 14019, 14021, 14022, 14193,
14206, 14216, 14217, 14222, 14223,
14224, 14225, 14226, 14227, 14234,
14244, 14251, 14262, 14263, 14274,
14291, 14334, 14335, 14342, 14355,
14370, 14380, 14394, 14424, 14433,
14467, 14489, 14507, 14524, 14541,
14542, 14544, 14545, 14550, 14565,
14598, 14627, 14628, 14629, 14630,
14631, 14644, 14688, 14761, 14828,
14830, 14831, 14841, 14870, 14873,
14874, 14885, 14905, 14968, 14969,
14970, 14982, 15021, 15022, 15023,
15024, 15030, 15033, 15035, 15045,
15062, 15074, 15081, 15328, 15332,
15334, 15338, 15345, 15346, 15356,
15357, 15360, 15452, 15522, 15533,
15545, 15575, 15582, 15593, 15609,
15616, 15677, 15734, 15744, 15746,
15756, 15774, 15775, 15807, 15810,
15819, 15821, 15823, 15837, 15851,
15875, 15895, 15896, 15909, 15925,
15941, 15957, 15973, 15989, 16003,
16011, 16017, 16028, 16031, 16034,
16043, 16053, 16055, 16061, 16068,
16071, 16080, 16088, 16109, 16142,
16143, 16170, 16172, 16186, 16187,
16206, 16217, 16226, 16229, 16273,
16288, 16303, 16318, 16333, 16348,
16351, 16353, 16370, 16415, 16642,
16678, 16679, 16689, 16730, 16731,
16755, 16782, 16783, 16786, 16788,
16789, 16794, 16806, 16825, 16830,
16838, 16841, 16873, 16883, 16884,
16894, 16916, 16931, 16949, 16957,

- 16960, 16988, 16996, 17012, 17084,
 17102, 17138, 17156, 17189, 17212,
 17218, 17292, 17293, 17398, 17399,
 17415, 17444, 17445, 17461, 17588,
 17589, 17611, 17678, 17690, 17691,
 17746, 17777, 17819, 17830, 17839,
 17848, 17891, 17901, 17911, 18025,
 18027, 18085, 18089, 18093, 18120,
 18131, 18149, 18150, 18151, 18158,
 18174, 18180, 18183, 18191, 18201,
 18216, 18224, 18232, 18243, 18259,
 18279, 18292, 18314, 18332, 18340,
 18342, 18345, 18352, 18357, 18434,
 18435, 18585, 18592, 18593, 18599,
 18602, 18605, 18612, 18613, 18616,
 18620, 18621, 18631, 18632, 18637,
 18638, 18650, 18658, 18667, 18668,
 18696, 18866, 18877, 18929, 18931,
 18938, 18941, 18942, 18946, 18950,
 18951, 18952, 18953, 18962, 18963,
 18967, 18970, 18971, 18972, 19020,
 19023, 19031, 19042, 19043, 19054,
 19055, 19063, 19075, 19076, 19086,
 19087, 19100, 19119, 19120, 19128,
 19129, 19180, 19190, 19216, 19230,
 19234, 19299, 19347, 19350, 19351,
 19366, 19369, 19392, 19459, 19460,
 19465, 19492, 19493, 19504, 19508,
 19542, 19547, 19555, 19589, 19596,
 19601, 19611, 19623, 19649, 19711,
 19740, 19779, 19786, 19797, 19871,
 19888, 19892, 19906, 19940, 20163,
 20214, 20230, 20254, 20278, 20287,
 20347, 20354, 20374, 20392, 20403,
 20405, 20435, 20438, 20463, 20562,
 20591, 20614, 20615, 20636, 20671,
 20697, 20770, 20831, 20843, 20906,
 20920, 20921, 20943, 20954, 20980,
 20998, 21052, 21054, 21069, 21071,
 21092, 21194, 21253, 21270, 21271,
 21310, 21311, 21315, 21316, 21338,
 21343, 21380, 21418, 21426, 21427,
 21431, 21432, 21833, 21835, 21841,
 23590, 24022, 24467, 24468, 24471,
 25422, 25429, 25434, 25462, 25468,
 25472, 25487, 25533, 25545, 25556,
 25564, 25614, 25620, 25621, 25636,
 25637, 25638, 25648, 25655, 25657
- file commands:
- \file_add_path:nN [9974](#)
 \g_file_curr_dir_str
 [148](#), [9654](#), [9817](#), [9823](#), [9840](#)
 \g_file_curr_ext_str
 [148](#), [9654](#), [9819](#), [9825](#), [9842](#)
- \g_file_curr_name_str
 [148](#), [8244](#), [9654](#), [9697](#),
[9818](#), [9824](#), [9841](#), [9952](#), [9956](#), [9958](#)
 \g_file_current_name_tl ... [544](#), [9951](#)
 \file_get_full_name:nN
 . [149](#), [289](#), [969](#), [9117](#), [9739](#), [9780](#),
[9793](#), [9974](#), [9977](#), [23952](#), [23963](#),
[23977](#), [24003](#), [24009](#), [24369](#), [24396](#)
 \file_get_md5five_hash:nN . [237](#), [23944](#)
 \file_get_size:nN [238](#), [23944](#)
 \file_get_timestamp:nN ... [238](#), [23944](#)
 \file_if_exist:nTF
 ... [149](#), [149](#), [149](#), [9778](#), [27163](#), [27165](#)
 \file_if_exist_input:n ... [238](#), [24001](#)
 \file_if_exist_input:nTF
 [238](#), [24001](#), [27162](#), [27164](#)
 \file_input:n
[149](#), [149](#), [238](#), [238](#), [9791](#), [27163](#), [27165](#)
 \file_input_stop: [238](#), [24014](#)
 \file_list: [9981](#)
 \file_log_list:
 [149](#), [545](#), [9883](#), [9981](#), [9982](#)
 \file_parse_full_name:nNNN
 [149](#), [9756](#), [9821](#), [9844](#)
 \file_path_include:n
 [237](#), [238](#), [238](#), [238](#), [9959](#)
 \file_path_remove:n [9959](#)
 \l_file_search_path_seq
 [149](#), [149](#), [9706](#),
[9743](#), [9960](#), [9964](#), [9965](#), [9968](#), [9972](#)
 \file_show_list: [149](#), [9883](#)
- file internal commands:
- \l__file_base_name_str
 [9701](#), [9741](#), [9773](#), [9787](#), [9789](#)
 \l__file_dir_str [9703](#), [9757](#), [9822](#), [9823](#)
 \l__file_ext_str
 [9703](#), [9757](#), [9758](#), [9822](#), [9825](#)
 \l__file_full_name_str
 [9701](#), [9752](#), [9754](#), [9756](#),
[9761](#), [9763](#), [9766](#), [9774](#), [9775](#), [9780](#),
[9781](#), [9793](#), [9794](#), [9796](#), [9963](#), [9964](#),
[9965](#), [9971](#), [9972](#), [23952](#), [23956](#),
[23963](#), [23969](#), [23977](#), [23981](#), [24003](#),
[24004](#), [24005](#), [24009](#), [24010](#), [24012](#)
 __file_get_details:nnN [23944](#)
 __file_get_full_name_search:nN [9739](#)
 __file_input:n .. [9791](#), [24005](#), [24012](#)
 __file_input_pop: [9791](#)
 __file_input_pop:nnn [9791](#)
 __file_input_push:n [9791](#)
 \g__file_internal_ior
 .. [9653](#), [9760](#), [9762](#), [9767](#), [9775](#), [9776](#)
 __file_list:N [9883](#)
 __file_list_aux:n [9883](#)

- __file_name_quote:nN ... [9710](#), [9772](#)
- __file_name_quote_aux:n . [9728](#), [9737](#)
- __file_name_sanitize_aux:n .. [9710](#)
- \l_file_name_str [9703](#), [9757](#), [9822](#), [9824](#)
- __file_parse_full_name_auxi:w [9844](#)
- __file_parse_full_name_split:nNNTF [9844](#)
- \g_file_record_seq ... [541](#), [543](#), [543](#), [9692](#), [9801](#), [9806](#), [9895](#), [9908](#), [9909](#)
- \g_file_stack_seq [541](#), [9668](#), [9815](#), [9832](#)
- __file_tmp:w [9671](#), [9675](#), [9679](#), [9685](#), [9689](#), [9864](#), [9879](#), [9881](#)
- \l_file_tmp_seq [9707](#), [9887](#), [9891](#), [9895](#), [9896](#), [9898](#), [9905](#), [9910](#)
- \l_file_tmp_tl . [9700](#), [9715](#), [9716](#), [9718](#), [9719](#), [9720](#), [9722](#), [9832](#), [9833](#)
- \finalhyphendemerits [367](#)
- \firstmark [368](#)
- \firstmarks [626](#)
- \firstvalidlanguage [908](#)
- flag commands:
 - \flag_clear:n [90](#), [90](#), [6165](#), [6180](#), [19809](#), [21276](#), [21277](#)
 - \flag_clear_new:n [90](#), [6179](#)
 - \flag_height:n [91](#), [6188](#), [6219](#), [6234](#), [21286](#), [21287](#), [21293](#), [21294](#)
 - \flag_if_exist:n [91](#)
 - \flag_if_exist:nTF [91](#), [6180](#), [6196](#), [6205](#)
 - \flag_if_exist_p:n [91](#), [6205](#)
 - \flag_if_raised:n [91](#)
 - \flag_if_raised:nTF . [91](#), [6210](#), [19817](#)
 - \flag_if_raised_p:n [91](#), [6210](#)
 - \flag_log:n [90](#), [6181](#)
 - \flag_new:n [90](#), [90](#), [434](#), [6160](#), [6180](#), [11984](#), [11985](#), [11986](#), [11987](#), [19799](#), [21180](#), [21181](#)
 - \flag_raise:n . [91](#), [6231](#), [21309](#), [21314](#)
 - \flag_raise_if_clear:n [238](#), [12018](#), [12027](#), [12035](#), [12052](#), [12061](#), [12092](#), [19835](#), [19857](#), [24016](#)
 - \flag_show:n [90](#), [6181](#)
- flag fp commands:
 - flag_fp_division_by_zero . [187](#), [11984](#)
 - flag_fp_invalid_operation [187](#), [11984](#)
 - flag_fp_overflow [187](#), [11984](#)
 - flag_fp_underflow [187](#), [11984](#)
- flag internal commands:
 - __flag_chk_exist:n [6191](#), [6210](#), [6219](#), [24016](#)
 - __flag_clear:wn [6165](#)
 - __flag_height_end:wn [6219](#)
 - __flag_height_loop:wn [6219](#)
- __flag_show:Nn [6181](#)
- \floatingpenalty [369](#)
- floor [192](#)
- \fmtname [145](#)
- \font [370](#)
- \fontchardp [627](#)
- \fontcharht [628](#)
- \fontcharic [629](#)
- \fontcharwd [630](#)
- \fontdimen [371](#)
- \fontid [909](#)
- \fontname [372](#)
- \forcecjktoken [1229](#)
- \formatname [910](#)
- fp commands:
 - \c_e_fp [186](#), [188](#), [13881](#)
 - \fp_abs:n [191](#), [196](#), [777](#), [17312](#), [22240](#), [22317](#), [22319](#), [22321](#), [23899](#), [23901](#)
 - \fp_add:Nn [180](#), [777](#), [13858](#)
 - \fp_compare:nNnTF [182](#), [183](#), [183](#), [184](#), [184](#), [13914](#), [14055](#), [14061](#), [14066](#), [14074](#), [14135](#), [14141](#), [17454](#), [17469](#), [17501](#), [22108](#), [22110](#), [22115](#), [22336](#), [22351](#), [22360](#), [22771](#), [23881](#), [25899](#), [26219](#), [26225](#), [26358](#), [26681](#), [26694](#), [26739](#)
 - \fp_compare:nTF [183](#), [184](#), [184](#), [184](#), [184](#), [190](#), [13898](#), [14027](#), [14033](#), [14038](#), [14046](#)
 - \fp_compare_p:n [183](#), [13898](#)
 - \fp_compare_p:nNn [182](#), [13914](#)
 - \fp_const:Nn [179](#), [13835](#), [13881](#), [13882](#), [13883](#), [13884](#)
 - \fp_do_until:n [184](#), [14024](#)
 - \fp_do_until:nNnn [183](#), [14052](#)
 - \fp_do_while:n [184](#), [14024](#)
 - \fp_do_while:nNnn [183](#), [14052](#)
 - \fp_eval:n [180](#), [183](#), [190](#), [190](#), [190](#), [190](#), [191](#), [191](#), [191](#), [191](#), [191](#), [191](#), [191](#), [191](#), [192](#), [192](#), [192](#), [193](#), [193](#), [193](#), [194](#), [194](#), [194](#), [195](#), [195](#), [196](#), [196](#), [667](#), [17309](#), [25734](#), [25735](#), [25739](#), [25743](#), [25804](#), [25805](#), [25806](#), [25807](#), [25816](#), [25824](#), [25825](#), [25826](#), [25892](#), [25901](#), [25914](#), [25915](#), [26081](#), [26098](#), [26099](#), [26108](#), [26109](#), [26113](#), [26115](#), [26119](#), [26124](#), [26140](#), [26141](#), [26213](#), [26230](#), [26245](#), [26247](#), [26351](#), [26360](#), [26371](#), [26372](#), [26609](#), [26626](#), [26627](#), [26635](#), [26636](#), [26641](#), [26643](#), [26647](#), [26652](#), [26666](#), [26667](#), [26683](#), [26688](#), [26689](#), [26696](#), [26706](#), [26707](#), [26708](#), [26709](#), [26718](#), [26719](#), [26720](#), [26721](#), [26730](#), [26731](#), [26732](#), [26733](#)

- 26754, 26755, 26837, 26852, 26853,
 27052, 27070, 27071, 27072, 27081,
 27092, 27093, 27094, 27121, 27122
 \fp_format:nn 196
 \fp_function:Nw 13737
 \fp_gadd:Nn 180, 13858
 .fp_gset:N 168, 11151
 \fp_gset:Nn .. 180, 13835, 13859, 13861
 \fp_gset_eq:NN 180, 13844, 13849
 \fp_gsub:Nn 180, 13858
 \fp_gzero:N 179, 13848, 13855
 \fp_gzero_new:N 179, 13852
 \fp_if_exist:NTF
 182, 13853, 13855, 13896
 \fp_if_exist_p:N 182, 13896
 \fp_if_nan:nTF 197
 \fp_log:N 187, 13868
 \fp_log:n 187, 13877
 \fp_max:nn 196, 17314
 \fp_min:nn 196, 17314
 \fp_new:N 179, 179,
 13832, 13853, 13855, 13885, 13886,
 13887, 13888, 22080, 22081, 22082,
 22202, 22203, 22418, 22419, 23706,
 23707, 23858, 23859, 26238, 26239
 \fp_new_function:Npn 13744
 .fp_set:N 168, 11151
 \fp_set:Nn 180, 13835, 13858,
 13860, 22096, 22097, 22098, 22209,
 22211, 22247, 22262, 22277, 22289,
 22291, 22304, 22305, 22334, 22335,
 22767, 22769, 23716, 23717, 23864,
 23866, 23893, 23894, 26218, 26221
 \fp_set_eq:NN .. 180, 13844, 13848,
 22252, 22267, 22279, 22337, 22338
 \fp_show:N 187, 13868
 \fp_show:n 187, 13877
 \fp_step_function:nnnN
 185, 14080, 14172
 \fp_step_inline:nnnn 185, 14150
 \fp_step_variable:nnnN .. 185, 14150
 \fp_sub:Nn 180, 13858
 \fp_to_decimal:N
 180, 181, 11977, 17115, 17146, 17309
 \fp_to_decimal:n 180, 180, 181, 181,
 181, 17115, 17311, 17313, 17315, 17317
 \fp_to_dim:N 181, 776, 17238
 \fp_to_dim:n 181, 186,
 17238, 22140, 22151, 22240, 22777,
 22808, 23795, 23803, 23909, 23911
 \fp_to_int:N 181, 17254
 \fp_to_int:n
 181, 781, 17254, 17470, 17472, 17535
 \fp_to_scientific:N
 181, 17061, 17092, 17099
 \fp_to_scientific:n . 181, 181, 17061
 \fp_to_tl:N .. 181, 11978, 13875, 17194
 \fp_to_tl:n 181,
 11587, 12017, 12026, 12051, 12060,
 12089, 13657, 13672, 13878, 13880,
 14107, 14108, 14127, 14138, 17194
 \fp_trap:nn 187, 187,
 608, 11988, 12103, 12104, 12105, 12106
 \fp_until_do:nn 184, 14024
 \fp_until_do:nNnn 184, 14052
 \fp_use:N
 181, 17309, 26224, 26228, 26233
 \fp_while_do:nn 184, 14024
 \fp_while_do:nNnn 184, 14052
 \fp_zero:N 179, 179, 13848, 13853, 26220
 \fp_zero_new:N 179, 13852
 \c_inf_fp 186, 195, 11598,
 13194, 14641, 14723, 15803, 15826,
 16033, 16036, 16040, 16064, 16355
 \c_nan_fp
 195, 611, 634, 11598, 12028, 12036,
 12108, 12314, 12333, 12339, 12520,
 12528, 12537, 12616, 12673, 12712,
 13106, 13183, 13195, 13659, 13674,
 13774, 14131, 16007, 17357, 17424
 \c_one_fp
 185, 661, 762, 13198, 13602, 13623,
 13881, 14231, 15066, 15797, 16002,
 16054, 16279, 16309, 16778, 17435
 \c_pi_fp .. 186, 195, 644, 13196, 13883
 \g_tmpa_fp 186, 13885
 \l_tmpa_fp 186, 13885
 \g_tmpb_fp 186, 13885
 \l_tmpb_fp 186, 13885
 \c_zero_fp ... 185, 666, 681, 11598,
 11645, 13199, 13614, 13626, 13833,
 13848, 13849, 14233, 14236, 14470,
 14637, 15806, 15827, 16030, 16067,
 16993, 17099, 17284, 22108, 22110,
 22115, 22351, 22360, 23881, 25899,
 26219, 26225, 26358, 26681, 26694
 fp internal commands:
 fp&_o:ww 668, 677, 14237
 fp&_tuple_o:ww 14237
 _fp*_o:ww 14602
 _fp*_tuple_o:ww 15092
 _fp+_o:ww 679, 680, 708, 14323
 _fp-_o:ww 679, 680, 14318
 _fp/_o:ww 688, 729, 14714
 _fp^_o:ww 15998
 _fp_acos_o:w 766, 769, 16934
 _fp_acot_o:Nw . 16254, 16256, 16766

- _fp_acotii_o:Nww [16776](#), [16779](#)
- _fp_acotii_o:ww [762](#)
- _fp_acsc_normal_o:NnwNnw
- [768](#), [16992](#), [17007](#), [17015](#)
- _fp_acsc_o:w [16986](#)
- _fp_add:NNNn [13858](#)
- _fp_add_big_i:wNww [682](#)
- _fp_add_big_i_o:wNww
- [679](#), [682](#), [14390](#), [14397](#)
- _fp_add_big_ii:wNww [682](#)
- _fp_add_big_ii_o:wNww [14393](#), [14397](#)
- _fp_add_inf_o:Nww [14339](#), [14359](#)
- _fp_add_normal_o:Nww
- [681](#), [14338](#), [14374](#)
- _fp_add_npos_o:NnwNnw
- [682](#), [14377](#), [14383](#)
- _fp_add_return_ii_o:Nww
- [14341](#), [14347](#), [14352](#)
- _fp_add_significand_carry_-
- o:wwwNN [683](#), [14430](#), [14445](#)
- _fp_add_significand_no_carry_-
- o:wwwNN [683](#), [14432](#), [14435](#)
- _fp_add_significand_o:NnnwnnnN
- [682](#), [683](#), [14400](#), [14408](#), [14413](#)
- _fp_add_significand_pack:NNNNNNN
- [14413](#)
- _fp_add_significand_test_o:N . [14413](#)
- _fp_add_zeros_o:Nww [14337](#), [14349](#)
- _fp_and_return:wNw [14237](#)
- _fp_array_count:n [11694](#),
- [12298](#), [13981](#), [13982](#), [15105](#), [17034](#)
- _fp_array_if_all_fp:nTF
- [11706](#), [13652](#)
- _fp_array_if_all_fp_loop:w . [11706](#)
- _fp_array_to_clist:n
- [12359](#), [17318](#), [17459](#)
- _fp_array_to_clist_loop:Nw . [17318](#)
- _fp_asec_o:w [16999](#)
- _fp_asin_auxi_o:NnNww
- [767](#), [769](#), [16964](#), [16967](#), [17026](#)
- _fp_asin_isqrt:wn [16967](#)
- _fp_asin_normal_o:NnwNnnnw
- [16925](#), [16941](#), [16952](#)
- _fp_asin_o:w [16919](#)
- _fp_atan_auxi:ww . [764](#), [16844](#), [16858](#)
- _fp_atan_auxii:w [16858](#)
- _fp_atan_combine_aux:ww [16885](#)
- _fp_atan_combine_o:NwwwwwN
- [763](#), [763](#), [16803](#), [16820](#), [16885](#)
- _fp_atan_default:w . [661](#), [762](#), [16766](#)
- _fp_atan_div:wnwnw
- [763](#), [16831](#), [16833](#)
- _fp_atan_inf_o:NNNw . [762](#), [16791](#),
- [16792](#), [16793](#), [16801](#), [16937](#), [17010](#)
- _fp_atan_near:wwn [16833](#)
- _fp_atan_near_aux:wwn [16833](#)
- _fp_atan_normal_o:NNnwNnw
- [762](#), [16795](#), [16811](#)
- _fp_atan_o:Nw [16258](#), [16260](#), [16766](#)
- _fp_atan_Taylor_break:w [16869](#)
- _fp_atan_Taylor_loop:www
- [764](#), [16864](#), [16869](#)
- _fp_atan_test_o:NnwNwN
- [768](#), [16814](#), [16818](#), [16974](#)
- _fp_atanii_o:Nww [16770](#), [16779](#)
- _fp_basics_pack_high:NNNNNw
- [683](#), [700](#), [11804](#), [14438](#), [14590](#),
- [14693](#), [14705](#), [14847](#), [15040](#), [15550](#)
- _fp_basics_pack_high_carry:w
- [600](#), [11804](#)
- _fp_basics_pack_low:NNNNNw
- [690](#), [700](#),
- [11804](#), [14440](#), [14592](#), [14695](#), [14707](#),
- [14849](#), [14989](#), [14991](#), [15042](#), [15552](#)
- _fp_basics_pack_weird_high:NNNNNNNw
- [197](#), [11815](#), [14449](#), [14858](#)
- _fp_basics_pack_weird_low:NNNNw
- [197](#), [11815](#), [14451](#), [14860](#)
- \c__fp_big_leading_shift_int
- [11790](#), [14919](#), [15238](#), [15248](#), [15258](#)
- \c__fp_big_middle_shift_int
- [11790](#), [14922](#), [14925](#), [14928](#),
- [14931](#), [14934](#), [14937](#), [14941](#), [15240](#),
- [15250](#), [15260](#), [15270](#), [15273](#), [15276](#)
- \c__fp_big_trailing_shift_int
- [11790](#), [14945](#), [15283](#)
- \c__fp_Bigg_leading_shift_int
- [11795](#), [14768](#), [14786](#)
- \c__fp_Bigg_middle_shift_int
- [11795](#), [14771](#), [14774](#), [14789](#), [14792](#)
- \c__fp_Bigg_trailing_shift_int
- [11795](#), [14777](#), [14795](#)
- _fp_binary_rev_type_o:Nww
- [13317](#), [15095](#), [15097](#)
- _fp_binary_type_o:Nww
- [13317](#), [15093](#), [15106](#)
- \c__fp_block_int [11603](#), [15502](#)
- _fp_case_return:nw [603](#),
- [11872](#), [11902](#), [11905](#), [11910](#), [12419](#),
- [15762](#), [16791](#), [16792](#), [16793](#), [17086](#),
- [17140](#), [17214](#), [17216](#), [17217](#), [17284](#)
- _fp_case_return_i_o:ww . [11879](#),
- [14340](#), [14354](#), [14363](#), [14635](#), [16782](#)
- _fp_case_return_ii_o:ww
- [11879](#), [14636](#), [16052](#), [16070](#), [16783](#)
- _fp_case_return_o:Nw
- [603](#), [604](#), [11873](#), [15797](#), [15802](#),

- 15805, 16002, 16007, 16030, 16033,
16036, 16279, 16309, 16993, 16995
- _fp_case_return_o:Nww
..... [11877](#), [14637](#), [14638](#),
[14641](#), [14642](#), [16054](#), [16063](#), [16066](#)
- _fp_case_return_same_o:w
..... [603](#), [604](#), [11875](#),
[14870](#), [14874](#), [15058](#), [15061](#), [15581](#),
[15809](#), [16027](#), [16264](#), [16272](#), [16287](#),
[16302](#), [16317](#), [16324](#), [16332](#), [16347](#),
[16922](#), [16930](#), [16948](#), [16994](#), [17011](#)
- _fp_case_use:nw
..... [603](#), [11871](#), [14365](#), [14633](#),
[14634](#), [14639](#), [14640](#), [14722](#), [14725](#),
[14872](#), [15574](#), [15577](#), [16038](#), [16265](#),
[16270](#), [16280](#), [16285](#), [16295](#), [16300](#),
[16310](#), [16315](#), [16325](#), [16330](#), [16340](#),
[16345](#), [16924](#), [16927](#), [16937](#), [16939](#),
[16945](#), [16989](#), [16991](#), [17002](#), [17005](#),
[17010](#), [17089](#), [17096](#), [17143](#), [17150](#)
- _fp_change_func_type:NNN
..... [11734](#), [13110](#), [15088](#),
[17071](#), [17125](#), [17202](#), [17248](#), [17263](#)
- _fp_change_func_type_aux:w . [11734](#)
- _fp_change_func_type_chk:NNN [11734](#)
- _fp_chk:w [590](#),
[592](#), [644](#), [680](#), [681](#), [682](#), [684](#), [690](#),
[692](#), [11588](#), [11598](#), [11599](#), [11600](#),
[11601](#), [11602](#), [11612](#), [11617](#), [11619](#),
[11620](#), [11641](#), [11644](#), [11646](#), [11656](#),
[11669](#), [11688](#), [11883](#), [11899](#), [12084](#),
[12089](#), [12316](#), [12362](#), [12371](#), [12373](#),
[13208](#), [13948](#), [13949](#), [14111](#), [14127](#),
[14131](#), [14195](#), [14196](#), [14199](#), [14210](#),
[14211](#), [14219](#), [14220](#), [14228](#), [14240](#),
[14243](#), [14247](#), [14250](#), [14324](#), [14344](#),
[14345](#), [14347](#), [14348](#), [14349](#), [14357](#),
[14360](#), [14371](#), [14372](#), [14374](#), [14383](#),
[14459](#), [14611](#), [14645](#), [14646](#), [14649](#),
[14730](#), [14868](#), [14876](#), [14878](#), [15055](#),
[15063](#), [15065](#), [15067](#), [15071](#), [15571](#),
[15583](#), [15585](#), [15794](#), [15811](#), [15813](#),
[15999](#), [16018](#), [16020](#), [16021](#), [16024](#),
[16041](#), [16044](#), [16047](#), [16072](#), [16073](#),
[16075](#), [16091](#), [16176](#), [16189](#), [16191](#),
[16195](#), [16199](#), [16261](#), [16274](#), [16276](#),
[16289](#), [16291](#), [16304](#), [16306](#), [16319](#),
[16321](#), [16334](#), [16336](#), [16349](#), [16359](#),
[16780](#), [16796](#), [16797](#), [16801](#), [16812](#),
[16919](#), [16932](#), [16934](#), [16950](#), [16953](#),
[16963](#), [16986](#), [16997](#), [16999](#), [17013](#),
[17015](#), [17020](#), [17082](#), [17103](#), [17106](#),
[17136](#), [17157](#), [17160](#), [17210](#), [17226](#),
[17229](#), [17305](#), [17306](#), [17436](#), [17438](#)
- _fp_compare:wNNNNw [13542](#)
- _fp_compare_aux:wn [13914](#)
- _fp_compare_back:ww . [13930](#), [14209](#)
- _fp_compare_back_any:ww .. [669](#),
[669](#), [670](#), [13617](#), [13927](#), [13930](#), [13998](#)
- _fp_compare_back_tuple:ww .. [13975](#)
- _fp_compare_nan:w [669](#), [13930](#)
- _fp_compare_npos:nwnw [668](#),
[669](#), [671](#), [13958](#), [14004](#), [14461](#), [15352](#)
- _fp_compare_return:w [13898](#)
- _fp_compare_significand:nnnnnnnn
..... [14004](#)
- _fp_cos_o:w [16276](#)
- _fp_cot_o:w [749](#), [16336](#)
- _fp_cot_zero_o:Nnw
..... [748](#), [749](#), [16294](#), [16336](#)
- _fp_csc_o:w [16291](#)
- _fp_decimate:nNnnnn
..... [601](#), [604](#), [745](#), [11825](#),
[11890](#), [11917](#), [12375](#), [14399](#), [14407](#),
[14486](#), [15840](#), [15844](#), [16214](#), [17166](#)
- _fp_decimate_:Nnnnn [11837](#)
- _fp_decimate_auxi:Nnnnn [602](#), [11841](#)
- _fp_decimate_auxii:Nnnnn ... [11841](#)
- _fp_decimate_auxiii:Nnnnn .. [11841](#)
- _fp_decimate_auxiv:Nnnnn ... [11841](#)
- _fp_decimate_auxix:Nnnnn ... [11841](#)
- _fp_decimate_auxv:Nnnnn [11841](#)
- _fp_decimate_auxvi:Nnnnn ... [11841](#)
- _fp_decimate_auxvii:Nnnnn .. [11841](#)
- _fp_decimate_auxviii:Nnnnn . [11841](#)
- _fp_decimate_auxix:Nnnnn [11841](#)
- _fp_decimate_auxxi:Nnnnn ... [11841](#)
- _fp_decimate_auxxii:Nnnnn .. [11841](#)
- _fp_decimate_auxxiii:Nnnnn . [11841](#)
- _fp_decimate_auxxiv:Nnnnn .. [11841](#)
- _fp_decimate_auxxv:Nnnnn ... [11841](#)
- _fp_decimate_auxxvi:Nnnnn .. [11841](#)
- _fp_decimate_pack:nnnnnnnnnw .
..... [602](#), [11848](#), [11867](#)
- _fp_decimate_pack:nnnnnnnw
..... [11868](#), [11869](#)
- _fp_decimate_tiny:Nnnnn [11837](#)
- _fp_div_npos_o:Nww
..... [692](#), [692](#), [14719](#), [14729](#)
- _fp_div_significand_calc:wnnnnnnnn
..... [695](#),
[696](#), [14746](#), [14755](#), [14803](#), [15654](#), [15661](#)
- _fp_div_significand_calc_
i:wnnnnnnnn [14755](#)
- _fp_div_significand_calc_
ii:wnnnnnnnn [14755](#)
- _fp_div_significand_i_o:wnnw ..
..... [692](#), [695](#), [14736](#), [14742](#)

- __fp_div_significand_ii:wwn ... [697](#), [14750](#), [14751](#), [14752](#), [14799](#)
- __fp_div_significand_iii:wwnnnnn ... [698](#), [14753](#), [14806](#)
- __fp_div_significand_iv:wwnnnnnn ... [698](#), [14809](#), [14814](#)
- __fp_div_significand_large_o:wwNNNNwN ... [700](#), [14840](#), [14854](#)
- __fp_div_significand_pack:NNN ... [699](#), [731](#), [14801](#), [14834](#), [15641](#), [15659](#), [15667](#)
- __fp_div_significand_small_o:wwNNNNwN ... [700](#), [14838](#), [14844](#)
- __fp_div_significand_test_o:w ... [699](#), [699](#), [14744](#), [14835](#)
- __fp_div_significand_v:NN ... [14819](#), [14821](#), [14824](#)
- __fp_div_significand_v:NNw ... [14814](#)
- __fp_div_significand_vi:Nw ... [698](#), [14814](#)
- __fp_division_by_zero_o:Nnw ... [608](#), [12048](#), [12096](#), [15578](#), [16355](#), [16356](#)
- __fp_division_by_zero_o:NNww ... [608](#), [12056](#), [12096](#), [14723](#), [14726](#), [16040](#)
- \c__fp_empty_tuple_fp ... [11689](#), [12514](#), [13169](#), [13179](#)
- __fp_ep_compare:www ... [15347](#), [16827](#)
- __fp_ep_compare_aux:www ... [15347](#)
- __fp_ep_div:wwwwn ... [760](#), [15377](#), [15488](#), [16756](#), [16843](#), [16847](#), [16856](#), [17023](#)
- __fp_ep_div_eps_pack:NNNNwN ... [15407](#)
- __fp_ep_div_epsilon:wnNNNNn ... [721](#)
- __fp_ep_div_epsilon:wnNNNNnn ... [15404](#), [15407](#)
- __fp_ep_div_epsilonii:wnNNNNnn ... [15407](#)
- __fp_ep_div_esti:wwwwn ... [721](#), [15383](#), [15386](#)
- __fp_ep_div_estii:wwnnwwn ... [15386](#)
- __fp_ep_div_estiii:NNNNNwwwn ... [15386](#)
- __fp_ep_inv_to_float_o:wN ... [750](#)
- __fp_ep_inv_to_float_o:wwN ... [758](#), [15484](#), [15492](#), [16298](#), [16313](#)
- __fp_ep_isqrt:wwn ... [15430](#), [16984](#)
- __fp_ep_isqrt_aux:wwn ... [15430](#)
- __fp_ep_isqrt_auxi:wwn ... [15433](#), [15435](#)
- __fp_ep_isqrt_auxii:wwnnwn ... [15430](#)
- __fp_ep_isqrt_epsilon:wN ... [724](#), [15467](#), [15470](#)
- __fp_ep_isqrt_epsilonii:wwN ... [15470](#)
- __fp_ep_isqrt_esti:wwnnwn ... [15445](#), [15448](#)
- __fp_ep_isqrt_estii:wwnnwn ... [15448](#)
- __fp_ep_isqrt_estiii:NNNNNwwwn ... [15448](#)
- __fp_ep_mul:wwwwn ... [15362](#), [16713](#), [16743](#), [16971](#), [16982](#)
- __fp_ep_mul_raw:wwwwn ... [15362](#), [16377](#), [16663](#)
- __fp_ep_to_ep:wwN ... [15313](#), [15364](#), [15367](#), [15379](#), [15382](#), [15432](#), [16972](#)
- __fp_ep_to_ep_end:www ... [15313](#)
- __fp_ep_to_ep_loop:N ... [757](#), [15313](#), [16664](#)
- __fp_ep_to_ep_zero:ww ... [15313](#)
- __fp_ep_to_fixed:wwn ... [15295](#), [16374](#), [16850](#), [16859](#), [16969](#)
- __fp_ep_to_fixed_auxi:www ... [15295](#)
- __fp_ep_to_fixed_auxii:nnnnnnnnwn ... [15295](#)
- __fp_ep_to_float_o:wN ... [750](#)
- __fp_ep_to_float_o:wwN ... [747](#), [758](#), [15484](#), [15496](#), [16268](#), [16283](#), [16762](#)
- __fp_error:nnnn ... [12017](#), [12025](#), [12034](#), [12051](#), [12059](#), [12087](#), [12110](#), [12309](#), [12311](#), [12332](#), [12337](#), [13105](#), [13655](#), [13670](#), [14107](#), [14126](#), [14137](#), [17077](#), [17131](#), [17205](#)
- __fp_exp_after_f:nw ... [597](#), [630](#), [12498](#)
- __fp_exp_after_any_f:Nnw ... [11759](#)
- __fp_exp_after_any_f:nw ... [598](#), [11759](#), [11785](#), [12500](#), [13274](#)
- __fp_exp_after_array_f:w ... [598](#), [11770](#), [13159](#), [14277](#), [14288](#), [14298](#), [14306](#)
- __fp_exp_after_f:nw ... [594](#), [630](#), [11646](#), [11764](#), [13207](#), [13345](#)
- __fp_exp_after_mark_f:nw ... [630](#), [12498](#)
- __fp_exp_after_normal:nNNw ... [11649](#), [11659](#), [11676](#)
- __fp_exp_after_normal:Nwwww ... [11678](#), [11686](#)
- __fp_exp_after_o:w ... [594](#), [11646](#), [11876](#), [11880](#), [11882](#), [12369](#), [12413](#), [12431](#), [13637](#), [14227](#), [14245](#), [14254](#), [14263](#), [14348](#), [15069](#), [16188](#), [16193](#), [17507](#)
- __fp_exp_after_special:nNNw ... [595](#), [11651](#), [11661](#), [11666](#)
- __fp_exp_after_stop_f:nw ... [11759](#)
- __fp_exp_after_tuple_f:nw ... [11770](#), [13446](#)
- __fp_exp_after_tuple_o:w ... [11770](#), [14252](#), [14255](#), [14258](#), [14260](#)
- __fp_exp_large:w ... [15887](#)
- __fp_exp_large:wN ... [15887](#)
- __fp_exp_large_after:wwn ... [15887](#)

- _fp_exp_large_i:wN [15887](#)
- _fp_exp_large_ii:wN [15887](#)
- _fp_exp_large_iii:wN [15887](#)
- _fp_exp_large_iv:wN [15887](#)
- _fp_exp_large_v:wN .. [15887](#), [16175](#)
- _fp_exp_normal_o:w .. [15799](#), [15813](#)
- _fp_exp_o:w [15559](#), [15794](#)
- _fp_exp_overflow:NN [15813](#)
- _fp_exp_pos_large:NnnNwn [15845](#), [15887](#)
- _fp_exp_pos_o:NNwnw [15816](#), [15818](#), [15821](#)
- _fp_exp_pos_o:Nnnwnw [15813](#)
- _fp_exp_Taylor:Nnnwn [15841](#), [15860](#), [15995](#)
- _fp_exp_Taylor_break:Nww ... [15860](#)
- _fp_exp_Taylor_ii:ww . [15866](#), [15869](#)
- _fp_exp_Taylor_loop:www [15860](#)
- _fp_expand:n [778](#), [11936](#), [17322](#)
- _fp_expand_loop:nwnN [11936](#)
- _fp_exponent:w [11620](#)
- \c_fp_five_int [12171](#),
[12195](#), [12208](#), [12221](#), [12228](#), [12281](#)
- _fp_fixed(calculation):wnw .. [709](#)
- _fp_fixed_add:nnNnnwnw [15188](#)
- _fp_fixed_add:Nnnnnwnw [15188](#)
- _fp_fixed_add:wnw
.. [710](#), [713](#), [15188](#), [15428](#), [15736](#),
[15744](#), [15755](#), [15773](#), [16855](#), [16915](#)
- _fp_fixed_add_after:NNNNNwn . [15188](#)
- _fp_fixed_add_one:wN
[710](#), [15120](#), [15421](#), [15877](#), [15886](#), [16981](#)
- _fp_fixed_add_pack:NNNNNwn . [15188](#)
- _fp_fixed_continue:wn
[15119](#), [15365](#), [15370](#), [15380](#), [15899](#),
[15915](#), [15931](#), [15947](#), [15963](#), [15979](#),
[16151](#), [16412](#), [16701](#), [16973](#), [16982](#)
- _fp_fixed_div_int:wnN [15157](#)
- _fp_fixed_div_int:wwN
.... [711](#), [15157](#), [15735](#), [15876](#), [16874](#)
- _fp_fixed_div_int_after:Nw ...
..... [712](#), [15157](#)
- _fp_fixed_div_int_auxi:wnw . [15157](#)
- _fp_fixed_div_int_auxii:wnw ...
..... [712](#), [15157](#)
- _fp_fixed_div_int_pack:Nw
..... [712](#), [15157](#)
- _fp_fixed_div_myriad:wn
..... [15125](#), [15425](#)
- _fp_fixed_inv_to_float_o:wN ...
..... [15491](#), [15818](#), [16087](#)
- _fp_fixed_mul:nnnnnnnw [15208](#)
- _fp_fixed_mul:wnw
.. [710](#), [711](#), [713](#), [756](#), [758](#), [15208](#),
[15374](#), [15405](#), [15420](#), [15422](#), [15426](#),
[15479](#), [15482](#), [15495](#), [15737](#), [15747](#),
[15787](#), [15878](#), [15896](#), [15996](#), [16097](#),
[16670](#), [16724](#), [16862](#), [16895](#), [16897](#)
- _fp_fixed_mul_add:nnnnwnnnn ...
..... [716](#), [15277](#), [15279](#)
- _fp_fixed_mul_add:nnnnwnnwN ...
..... [717](#), [15284](#), [15290](#)
- _fp_fixed_mul_add:Nwnnnwnnnn ...
.... [716](#), [15241](#), [15251](#), [15262](#), [15266](#)
- _fp_fixed_mul_add:www . [715](#), [15235](#)
- _fp_fixed_mul_after:wnw
..... [714](#), [15127](#), [15133](#), [15136](#),
[15210](#), [15237](#), [15247](#), [15257](#), [16114](#)
- _fp_fixed_mul_one_minus_-
mul:wnw [15235](#)
- _fp_fixed_mul_short:wnw
..... [711](#), [15134](#),
[15403](#), [15424](#), [15466](#), [15468](#), [16908](#)
- _fp_fixed_mul_sub_back:www ...
..... [715](#), [15235](#),
[15480](#), [16691](#), [16693](#), [16694](#), [16695](#),
[16696](#), [16697](#), [16698](#), [16699](#), [16700](#),
[16704](#), [16706](#), [16707](#), [16708](#), [16709](#),
[16710](#), [16711](#), [16712](#), [16737](#), [16739](#),
[16740](#), [16741](#), [16742](#), [16745](#), [16747](#),
[16748](#), [16749](#), [16750](#), [16875](#), [16883](#)
- _fp_fixed_one_minus_mul:wnw ...
..... [715](#), [716](#), [15255](#)
- _fp_fixed_sub:wnw
.... [15188](#), [15472](#), [15753](#), [15769](#),
[15781](#), [16416](#), [16856](#), [16913](#), [16979](#)
- _fp_fixed_to_float_o:Nw
..... [15498](#), [15762](#)
- _fp_fixed_to_float_o:wN
.... [710](#), [725](#), [765](#), [15485](#), [15498](#),
[15782](#), [15792](#), [15816](#), [16083](#), [16903](#)
- _fp_fixed_to_float_pack:ww ...
..... [15531](#), [15541](#)
- _fp_fixed_to_float_rad_o:wN ...
..... [15493](#), [16903](#)
- _fp_fixed_to_float_round_-
up:wnnnnw [15544](#), [15548](#)
- _fp_fixed_to_float_zero:w
..... [15527](#), [15536](#)
- _fp_fixed_to_loop:N
..... [15504](#), [15514](#), [15518](#)
- _fp_fixed_to_loop_end:w
..... [15520](#), [15524](#)
- _fp_from_dim:wNNnnnnnn [17273](#)
- _fp_from_dim:wnnnnwNn [17301](#), [17302](#)
- _fp_from_dim:wnnnnwNw [17273](#)
- _fp_from_dim:wNw [17273](#)

__fp_from_dim_test:ww
 [776](#), [12592](#), [12629](#), [13226](#), [17273](#)
 __fp_func_to_name:N
 [11964](#), [13105](#), [13114](#)
 __fp_func_to_name_aux:w [11964](#)
 __fp_function_apply:nw
 .. [663](#), [663](#), [664](#), [13739](#), [13755](#), [13777](#)
 __fp_function_args:Nwn .. [663](#), [13744](#)
 __fp_function_store:wwNwnn
 [664](#), [13777](#)
 __fp_function_store_end:wnnn ...
 [664](#), [13777](#)
 \c__fp_half_prec_int
 [11603](#), [12833](#), [12865](#)
 __fp_if_type_fp:NTwFw [596](#),
 [661](#), [11705](#), [11713](#), [11720](#), [11736](#),
 [11763](#), [13664](#), [13678](#), [13906](#), [13932](#),
 [13933](#), [14100](#), [14101](#), [14102](#), [14268](#)
 __fp_inf_fp:N [11616](#), [12072](#)
 __fp_int:wTF [11883](#), [17438](#)
 __fp_int_eval:w
 [599](#), [614](#), [615](#), [616](#), [629](#),
 [644](#), [682](#), [690](#), [690](#), [693](#), [697](#), [725](#),
 [737](#), [11573](#), [11630](#), [11698](#), [11829](#),
 [11832](#), [12245](#), [12249](#), [12261](#), [12262](#),
 [12298](#), [12381](#), [12385](#), [12424](#), [12639](#),
 [12644](#), [12686](#), [12775](#), [12786](#), [12835](#),
 [12866](#), [12872](#), [12873](#), [12919](#), [12929](#),
 [12931](#), [12947](#), [12949](#), [12972](#), [12974](#),
 [13140](#), [13362](#), [13577](#), [13919](#), [14387](#),
 [14395](#), [14416](#), [14418](#), [14439](#), [14441](#),
 [14450](#), [14452](#), [14481](#), [14487](#), [14497](#),
 [14499](#), [14573](#), [14575](#), [14591](#), [14593](#),
 [14597](#), [14613](#), [14653](#), [14661](#), [14663](#),
 [14665](#), [14667](#), [14670](#), [14673](#), [14675](#),
 [14694](#), [14696](#), [14706](#), [14708](#), [14734](#),
 [14737](#), [14745](#), [14747](#), [14768](#), [14771](#),
 [14774](#), [14777](#), [14786](#), [14789](#), [14792](#),
 [14795](#), [14802](#), [14804](#), [14810](#), [14818](#),
 [14820](#), [14822](#), [14828](#), [14848](#), [14850](#),
 [14859](#), [14861](#), [14882](#), [14903](#), [14907](#),
 [14919](#), [14922](#), [14925](#), [14928](#), [14931](#),
 [14934](#), [14937](#), [14940](#), [14944](#), [14956](#),
 [14960](#), [14964](#), [14967](#), [14988](#), [14990](#),
 [14992](#), [15002](#), [15041](#), [15043](#), [15052](#),
 [15123](#), [15128](#), [15130](#), [15137](#), [15140](#),
 [15143](#), [15146](#), [15149](#), [15152](#), [15161](#),
 [15173](#), [15181](#), [15183](#), [15193](#), [15195](#),
 [15202](#), [15211](#), [15213](#), [15216](#), [15219](#),
 [15222](#), [15225](#), [15238](#), [15240](#), [15248](#),
 [15250](#), [15258](#), [15260](#), [15270](#), [15273](#),
 [15276](#), [15283](#), [15298](#), [15316](#), [15319](#),
 [15375](#), [15389](#), [15391](#), [15397](#), [15410](#),
 [15412](#), [15414](#), [15438](#), [15454](#), [15461](#),
 [15462](#), [15485](#), [15502](#), [15506](#), [15551](#),
 [15553](#), [15595](#), [15606](#), [15625](#), [15627](#),
 [15629](#), [15642](#), [15655](#), [15660](#), [15662](#),
 [15668](#), [15685](#), [15686](#), [15687](#), [15688](#),
 [15689](#), [15690](#), [15695](#), [15697](#), [15699](#),
 [15701](#), [15703](#), [15708](#), [15710](#), [15712](#),
 [15714](#), [15716](#), [15718](#), [15740](#), [15748](#),
 [15832](#), [15881](#), [16094](#), [16115](#), [16117](#),
 [16120](#), [16123](#), [16126](#), [16129](#), [16145](#),
 [16171](#), [16181](#), [16197](#), [16364](#), [16396](#),
 [16405](#), [16587](#), [16588](#), [16611](#), [16621](#),
 [16630](#), [16648](#), [16657](#), [16664](#), [16675](#),
 [16685](#), [16718](#), [16728](#), [16753](#), [16762](#),
 [16805](#), [16822](#), [16824](#), [16836](#), [16837](#),
 [16878](#), [16889](#), [16900](#), [16958](#), [17110](#),
 [17233](#), [17287](#), [17389](#), [17395](#), [17411](#)
 __fp_int_eval_end: [11573](#),
 [11630](#), [11701](#), [11820](#), [12298](#), [12395](#),
 [12399](#), [13578](#), [13919](#), [14597](#), [14632](#),
 [14824](#), [15183](#), [15319](#), [16145](#), [16197](#),
 [16397](#), [16406](#), [16675](#), [16685](#), [16728](#),
 [16753](#), [16837](#), [17390](#), [17396](#), [17412](#)
 __fp_int_p:w [11883](#)
 __fp_int_to_roman:w [11573](#),
 [11832](#), [12847](#), [12879](#), [15622](#), [15890](#)
 __fp_invalid_operation:nw
 . [608](#), [12014](#), [12096](#), [12108](#), [17091](#),
 [17098](#), [17145](#), [17152](#), [17252](#), [17267](#)
 __fp_invalid_operation_o:nw ...
 [608](#), [12107](#), [13114](#),
 [14872](#), [15082](#), [15574](#), [16271](#), [16286](#),
 [16301](#), [16316](#), [16331](#), [16346](#), [16928](#),
 [16946](#), [16962](#), [16990](#), [17003](#), [17019](#)
 __fp_invalid_operation_o:Nww ...
 [608](#), [12022](#), [12096](#),
 [13315](#), [14367](#), [14639](#), [14640](#), [16182](#)
 __fp_invalid_operation_o:nww . [15107](#)
 __fp_invalid_operation_tl_o:nn .
 [608](#), [12031](#), [12096](#), [12357](#), [17458](#)
 \c__fp_leading_shift_int
 [11786](#), [15128](#),
 [15137](#), [15211](#), [16115](#), [16611](#), [16648](#)
 __fp_ln_c:NwNw [733](#), [733](#), [15719](#), [15750](#)
 __fp_ln_div_after:Nw
 [731](#), [15621](#), [15670](#)
 __fp_ln_div_i:w [15643](#), [15652](#)
 __fp_ln_div_ii:wnn
 .. [15646](#), [15647](#), [15648](#), [15649](#), [15657](#)
 __fp_ln_div_vi:wnn ... [15650](#), [15665](#)
 __fp_ln_exponent:wn [734](#), [15597](#), [15759](#)
 __fp_ln_exponent_one:ww [15764](#), [15778](#)
 __fp_ln_exponent_small:NNww ...
 [15767](#), [15771](#), [15784](#)
 \c__fp_ln_i_fixed_tl [15562](#)

- \c__fp_ln_ii_fixed_tl [15562](#)
- \c__fp_ln_iii_fixed_tl [15562](#)
- \c__fp_ln_iv_fixed_tl [15562](#)
- \c__fp_ln_ix_fixed_tl [15562](#)
- __fp_ln_npos_o:w
 - [727](#), [728](#), [15583](#), [15585](#)
- __fp_ln_o:w .. [727](#), [742](#), [15561](#), [15571](#)
- __fp_ln_significand:NNNNnnnN ...
 - [728](#), [15596](#), [15599](#), [16095](#)
- __fp_ln_square_t_after:w
 - [15694](#), [15726](#)
- __fp_ln_square_t_pack:NNNNnw ...
 - .. [15696](#), [15698](#), [15700](#), [15702](#), [15724](#)
- __fp_ln_t_large:NNw
 - [732](#), [15675](#), [15682](#), [15692](#)
- __fp_ln_t_small:Nw ... [15673](#), [15680](#)
- __fp_ln_t_small:w [732](#)
- __fp_ln_Taylor:wwNw [733](#), [15727](#), [15728](#)
- __fp_ln_Taylor_break:w [15733](#), [15744](#)
- __fp_ln_Taylor_loop:www
 - [15729](#), [15730](#), [15739](#)
- __fp_ln_twice_t_after:w [15707](#), [15723](#)
- __fp_ln_twice_t_pack:Nw . [15709](#),
 - [15711](#), [15713](#), [15715](#), [15717](#), [15722](#)
- \c__fp_ln_vi_fixed_tl [15562](#)
- \c__fp_ln_vii_fixed_tl [15562](#)
- \c__fp_ln_viii_fixed_tl [15562](#)
- \c__fp_ln_x_fixed_tl
 - [15562](#), [15781](#), [15788](#)
- __fp_ln_x_ii:wnnnn ... [15601](#), [15619](#)
- __fp_ln_x_iii:NNNNNNw . [15628](#), [15632](#)
- __fp_ln_x_iii_var:NNNNnw
 - [15626](#), [15634](#)
- __fp_ln_x_iv:wnnnnnnnn
 - [730](#), [15624](#), [15639](#)
- \c__fp_max_exp_exponent_int
 - [11609](#), [15824](#)
- \c__fp_max_exponent_int .. [11607](#),
 - [11613](#), [11634](#), [15336](#), [15538](#), [16150](#)
- \c__fp_middle_shift_int
 - [11786](#), [15140](#),
[15143](#), [15146](#), [15149](#), [15213](#), [15216](#),
[15219](#), [15222](#), [16117](#), [16120](#), [16123](#),
[16126](#), [16614](#), [16621](#), [16651](#), [16657](#)
- __fp_minmax_aux_o:Nw [14181](#)
- __fp_minmax_auxi:ww
 - [14203](#), [14215](#), [14222](#)
- __fp_minmax_auxii:ww
 - [14205](#), [14213](#), [14222](#)
- __fp_minmax_break_o:w . [14196](#), [14226](#)
- __fp_minmax_loop:Nww
 - [675](#), [14190](#), [14192](#), [14198](#)
- __fp_minmax_o:Nw
 - [668](#), [13893](#), [13895](#), [14181](#)
- \c__fp_minus_min_exponent_int ...
 - [11607](#), [11635](#)
- __fp_misused:n . [11586](#), [11590](#), [11691](#)
- __fp_mul_cases_o:NnNnw
 - [692](#), [14604](#), [14610](#), [14716](#)
- __fp_mul_cases_o:nNnnnw [14610](#)
- __fp_mul_npos_o:Nww
 - [688](#), [690](#), [692](#), [776](#), [14607](#), [14648](#), [17304](#)
- __fp_mul_significand_drop:NNNNnw
 - [690](#), [14657](#)
- __fp_mul_significand_keep:NNNNnw
 - [14657](#)
- __fp_mul_significand_large_-
f:NwwNNNN [14687](#), [14691](#)
- __fp_mul_significand_o:nnnnNnnnn
 - [690](#), [690](#), [14655](#), [14657](#)
- __fp_mul_significand_small_-
f:NNwwN [14685](#), [14702](#)
- __fp_mul_significand_test_f:NNN
 - [691](#), [14659](#), [14682](#)
- \c__fp_myriad_int [11606](#),
 - [15123](#), [15154](#), [15155](#), [15232](#), [15293](#)
- __fp_neg_sign:N
 - [680](#), [11629](#), [14321](#), [14474](#)
- __fp_new_function:NNnnn [13744](#)
- __fp_not_o:w [668](#), [13133](#), [14228](#)
- \c__fp_one_fixed_tl
 - [15117](#), [15735](#), [15891](#),
[16151](#), [16175](#), [16807](#), [16874](#), [16979](#)
- __fp_overflow:w
 - .. [594](#), [608](#), [610](#), [11637](#), [12096](#), [15826](#)
- \c__fp_overflowing_fp
 - [11610](#), [17092](#), [17146](#)
- __fp_pack:NNNNnw .. [11786](#), [15129](#),
 - [15139](#), [15142](#), [15145](#), [15148](#), [15151](#),
[15212](#), [15215](#), [15218](#), [15221](#), [15224](#),
[16116](#), [16119](#), [16122](#), [16125](#), [16128](#)
- __fp_pack_big:NNNNNNw ... [11790](#),
 - [14921](#), [14924](#), [14927](#), [14930](#), [14933](#),
[14936](#), [14939](#), [14943](#), [15239](#), [15249](#),
[15259](#), [15269](#), [15272](#), [15275](#), [15282](#)
- __fp_pack_Bigg:NNNNNNw
 - [11795](#), [14770](#),
[14773](#), [14776](#), [14788](#), [14791](#), [14794](#)
- __fp_pack_eight:wNNNNNNNN
 - [600](#), [686](#), [11802](#),
[14583](#), [14892](#), [15304](#), [16383](#), [16384](#)
- __fp_pack_twice_four:wNNNNNNNN .
 - [600](#),
[11800](#), [12406](#), [12407](#), [14525](#), [14526](#),
[15305](#), [15306](#), [15307](#), [15339](#), [15340](#),
[15341](#), [15529](#), [15530](#), [15863](#), [15864](#),
[15865](#), [16385](#), [16386](#), [16599](#), [17297](#)

__fp_parse:n
 [620](#), [632](#), [644](#), [652](#), [664](#), [666](#),
 [667](#), [673](#), [777](#), [12437](#), [12589](#), [13250](#),
 [13779](#), [13836](#), [13838](#), [13840](#), [13863](#),
 [13901](#), [13918](#), [13928](#), [14085](#), [14145](#),
 [17067](#), [17121](#), [17199](#), [17244](#), [17259](#),
 [17313](#), [17315](#), [17317](#), [17456](#), [17496](#)
 __fp_parse_after:ww [13250](#)
 __fp_parse_apply_binary:NwNwN ..
 [624](#), [625](#), [628](#), [628](#), [656](#), [13288](#), [13456](#)
 __fp_parse_apply_binary_chk:NN ..
 [13288](#), [13319](#), [13332](#)
 __fp_parse_apply_binary_-
 error:NNN [13288](#)
 __fp_parse_apply_comma:NwNwN ...
 [656](#), [13415](#)
 __fp_parse_apply_compare:NwNNNNwN
 [13601](#), [13610](#)
 __fp_parse_apply_compare_-
 aux:NNwN [13622](#), [13625](#), [13630](#)
 __fp_parse_apply_function:NNNwN
 [647](#), [13082](#), [13243](#)
 __fp_parse_apply_unary:NNNwN ...
 [13087](#), [13119](#), [13234](#)
 __fp_parse_apply_unary_chk:nNNNw
 [13098](#), [13099](#), [13102](#)
 __fp_parse_apply_unary_chk:nNNw
 [13087](#)
 __fp_parse_apply_unary_chk:NwNw
 [13087](#)
 __fp_parse_apply_unary_error:NNw
 [13087](#), [15089](#)
 __fp_parse_apply_unary_type:NNN
 [13087](#)
 __fp_parse_caseless_inf:N ... [13200](#)
 __fp_parse_caseless_infinity:N ..
 [13200](#)
 __fp_parse_caseless_nan:N ... [13200](#)
 __fp_parse_compare:NNNNNNN .. [13542](#)
 __fp_parse_compare_auxi:NNNNNNN
 [13542](#)
 __fp_parse_compare_auxii:NNNNN ..
 [13542](#)
 __fp_parse_compare_end:NNNNw .. [13542](#)
 __fp_parse_continue:NwN
 [624](#), [625](#), [652](#), [13277](#), [13290](#),
 [13443](#), [13640](#), [14285](#), [14295](#), [14303](#)
 __fp_parse_continue_compare:NNwNN
 [13633](#), [13648](#)
 __fp_parse_digits:N [12454](#)
 __fp_parse_digits_i:N [12454](#)
 __fp_parse_digits_ii:N [12454](#)
 __fp_parse_digits_iii:N [12454](#)
 __fp_parse_digits_iv:N [12454](#)
 __fp_parse_digits_v:N [12454](#)
 __fp_parse_digits_vi:N
 [12454](#), [12791](#), [12839](#)
 __fp_parse_digits_vii:N
 [638](#), [12454](#), [12778](#), [12828](#)
 __fp_parse_excl_error: [13542](#)
 __fp_parse_expand:w [628](#), [628](#), [629](#),
 [629](#), [12451](#), [12453](#), [12463](#), [12503](#),
 [12565](#), [12609](#), [12618](#), [12621](#), [12625](#),
 [12662](#), [12696](#), [12734](#), [12736](#), [12755](#),
 [12757](#), [12779](#), [12796](#), [12809](#), [12829](#),
 [12859](#), [12887](#), [12903](#), [12914](#), [12937](#),
 [12966](#), [12976](#), [12983](#), [12996](#), [13012](#),
 [13032](#), [13043](#), [13129](#), [13152](#), [13164](#),
 [13239](#), [13248](#), [13256](#), [13269](#), [13392](#),
 [13410](#), [13434](#), [13460](#), [13506](#), [13526](#),
 [13595](#), [13608](#), [13742](#), [13762](#), [14281](#)
 __fp_parse_exponent:N
 [642](#), [12564](#), [12770](#), [12919](#), [12986](#), [12988](#)
 __fp_parse_exponent:Nw
 [12794](#), [12807](#),
 [12856](#), [12884](#), [12935](#), [12964](#), [12983](#)
 __fp_parse_exponent_aux:N ... [12988](#)
 __fp_parse_exponent_body:N
 [13014](#), [13018](#)
 __fp_parse_exponent_digits:N ...
 [13022](#), [13034](#)
 __fp_parse_exponent_keep:N .. [13045](#)
 __fp_parse_exponent_keep:NTF ...
 [13025](#), [13045](#)
 __fp_parse_exponent_sign:N
 [13004](#), [13008](#)
 __fp_parse_function:NNN
 [12157](#), [12159](#), [12161](#),
 [12164](#), [13232](#), [13893](#), [13895](#), [16254](#),
 [16256](#), [16258](#), [16260](#), [17347](#), [17349](#)
 __fp_parse_function_all_fp_-
 o:nnw [12291](#), [13650](#), [14183](#)
 __fp_parse_function_one_two:nnw
 [762](#), [13662](#), [16768](#), [16774](#), [17431](#)
 __fp_parse_function_one_two_-
 aux:nnw [13662](#)
 __fp_parse_function_one_two_-
 auxii:nnw [13662](#)
 __fp_parse_function_one_two_-
 error_o:w [13662](#)
 __fp_parse_infix:NN
 .. [630](#), [634](#), [650](#), [655](#), [664](#), [12502](#),
 [12674](#), [12713](#), [13161](#), [13187](#), [13192](#),
 [13207](#), [13229](#), [13345](#), [13348](#), [13408](#)
 __fp_parse_infix_!:N [13542](#)
 __fp_parse_infix_&:Nw [13499](#)
 __fp_parse_infix(:N [13482](#)
 __fp_parse_infix_):N [13398](#)

- _fp_parse_infix_*:N [13484](#)
- _fp_parse_infix_+:N [628](#), [664](#), [12451](#), [13450](#)
- _fp_parse_infix_,:N [13415](#)
- _fp_parse_infix_-:N [13450](#)
- _fp_parse_infix_/:N [13450](#)
- _fp_parse_infix_::N . [13516](#), [14266](#)
- _fp_parse_infix_<:N [13542](#)
- _fp_parse_infix_=:N [13542](#)
- _fp_parse_infix_>:N [13542](#)
- _fp_parse_infix_?:N [13516](#)
- _fp_parse_infix_⟨operation₂⟩:N [628](#)
- _fp_parse_infix_^:N [13450](#)
- _fp_parse_infix_after_operand:NwN
.... [634](#), [12557](#), [12635](#), [13136](#), [13343](#)
- _fp_parse_infix_and:N [13450](#), [13515](#)
- _fp_parse_infix_check:NNN
..... [13368](#), [13378](#)
- _fp_parse_infix_comma:w [656](#), [13415](#)
- _fp_parse_infix_end:N
..... [652](#), [655](#), [13257](#), [13262](#), [13270](#), [13396](#)
- _fp_parse_infix_mark:NNN
..... [13355](#), [13395](#)
- _fp_parse_infix_mul:N
..... [657](#), [13358](#), [13366](#), [13450](#), [13483](#), [13492](#)
- _fp_parse_infix_or:N . [13450](#), [13514](#)
- _fp_parse_infix_|:Nw [13499](#)
- _fp_parse_large:N [636](#), [12741](#), [12824](#)
- _fp_parse_large_leading:wwNN ..
..... [640](#), [12826](#), [12831](#)
- _fp_parse_large_round:NN
..... [640](#), [12867](#), [12939](#)
- _fp_parse_large_round_aux:wNN .
..... [12939](#)
- _fp_parse_large_round_test:NN .
..... [12939](#)
- _fp_parse_large_trailing:wwNN .
..... [640](#), [12837](#), [12861](#)
- _fp_parse_letters:N
..... [634](#), [634](#), [12650](#), [12664](#)
- _fp_parse_lparen_after:NwN . [13142](#)
- _fp_parse_o:n
..... [620](#), [13250](#), [14083](#), [14084](#), [17428](#), [17491](#)
- _fp_parse_one:Nw
..... [623-627](#), [628](#), [635](#), [650](#),
..... [652](#), [12451](#), [12474](#), [12718](#), [13081](#), [13283](#)
- _fp_parse_one_digit:NN
..... [648](#), [12490](#), [12633](#)
- _fp_parse_one_fp:NN
..... [630](#), [12482](#), [12498](#)
- _fp_parse_one_other:NN [12493](#), [12641](#)
- _fp_parse_one_register:NN
..... [12485](#), [12555](#)
- _fp_parse_one_register_aux:Nw .
..... [12555](#)
- _fp_parse_one_register_-
auxii:wwNw [12555](#)
- _fp_parse_one_register_dim:ww .
..... [12555](#)
- _fp_parse_one_register_int:www
..... [12555](#)
- _fp_parse_one_register_-
math:NNw [12596](#)
- _fp_parse_one_register_mu:www .
..... [12555](#)
- _fp_parse_one_register_-
special:N [12560](#), [12596](#)
- _fp_parse_one_register_wd:Nw [12596](#)
- _fp_parse_one_register_wd:w . [12596](#)
- _fp_parse_operand:Nw
..... [623-626](#), [628](#),
..... [652](#), [656](#), [663](#), [663](#), [12451](#), [13125](#),
..... [13127](#), [13148](#), [13150](#), [13239](#), [13248](#),
..... [13255](#), [13268](#), [13277](#), [13433](#), [13459](#),
..... [13525](#), [13608](#), [13742](#), [13762](#), [14280](#)
- _fp_parse_pack_carry:w . [639](#), [12811](#)
- _fp_parse_pack_leading:NNNNnw
..... [12774](#), [12811](#), [12834](#)
- _fp_parse_pack_trailing:NNNNnw
.. [12784](#), [12811](#), [12853](#), [12864](#), [12871](#)
- _fp_parse_prefix:NNN . [12653](#), [12698](#)
- _fp_parse_prefix!:Nw [13115](#)
- _fp_parse_prefix(:Nw [13142](#)
- _fp_parse_prefix):Nw [13174](#)
- _fp_parse_prefix+:Nw [13081](#)
- _fp_parse_prefix-:Nw [13115](#)
- _fp_parse_prefix.:Nw [13134](#)
- _fp_parse_prefix_unknown:NNN [12698](#)
- _fp_parse_return_semicolon:w ..
..... [12452](#), [12461](#), [12694](#),
..... [12901](#), [12912](#), [12994](#), [13026](#), [13041](#)
- _fp_parse_round:Nw [12162](#)
- _fp_parse_round_after:wN
..... [642](#), [12916](#), [12921](#), [12971](#)
- _fp_parse_round_loop:N ... [642](#),
..... [642](#), [643](#), [12889](#), [12932](#), [12950](#), [12975](#)
- _fp_parse_round_up:N [12889](#)
- _fp_parse_small:N [637](#), [12761](#), [12772](#)
- _fp_parse_small_leading:wwNN ..
..... [638](#), [12776](#), [12781](#), [12843](#)
- _fp_parse_small_round:NN
..... [12803](#), [12921](#), [12960](#)
- _fp_parse_small_trailing:wwNN .
..... [638](#), [12789](#), [12798](#), [12875](#)
- _fp_parse_strim_end:w [12747](#)
- _fp_parse_strim_zeros:N
..... [636](#), [648](#), [12728](#), [12747](#), [13140](#)

- __fp_parse_trim_end:w [12721](#)
- __fp_parse_trim_zeros:N [12639](#), [12721](#)
- __fp_parse_unary_function:NNN ..
..... [13232](#), [14313](#), [14315](#),
[14317](#), [15559](#), [15561](#), [16242](#), [16248](#)
- __fp_parse_word:Nw [634](#), [12647](#), [12664](#)
- __fp_parse_word_abs:N [14312](#)
- __fp_parse_word_acos:N [16234](#)
- __fp_parse_word_acosd:N [16234](#)
- __fp_parse_word_acot:N [16253](#)
- __fp_parse_word_acotd:N [16253](#)
- __fp_parse_word_acsc:N [16234](#)
- __fp_parse_word_acscd:N [16234](#)
- __fp_parse_word_asec:N [16234](#)
- __fp_parse_word_asecd:N [16234](#)
- __fp_parse_word_asin:N [16234](#)
- __fp_parse_word_asind:N [16234](#)
- __fp_parse_word_atan:N [16253](#)
- __fp_parse_word_atand:N [16253](#)
- __fp_parse_word_bp:N [13203](#)
- __fp_parse_word_cc:N [13203](#)
- __fp_parse_word_ceil:N [12156](#)
- __fp_parse_word_cm:N [13203](#)
- __fp_parse_word_cos:N [16234](#)
- __fp_parse_word_cosd:N [16234](#)
- __fp_parse_word_cot:N [16234](#)
- __fp_parse_word_cotd:N [16234](#)
- __fp_parse_word_csc:N [16234](#)
- __fp_parse_word_cscd:N [16234](#)
- __fp_parse_word_dd:N [13203](#)
- __fp_parse_word_deg:N [13189](#)
- __fp_parse_word_em:N [13222](#)
- __fp_parse_word_ex:N [13222](#)
- __fp_parse_word_exp:N [15558](#)
- __fp_parse_word_false:N [13189](#)
- __fp_parse_word_floor:N [12156](#)
- __fp_parse_word_in:N [13203](#)
- __fp_parse_word_inf:N
..... [13189](#), [13200](#), [13201](#)
- __fp_parse_word_ln:N [15558](#)
- __fp_parse_word_max:N [13892](#)
- __fp_parse_word_min:N [13892](#)
- __fp_parse_word_mm:N [13203](#)
- __fp_parse_word_nan:N . [13189](#), [13202](#)
- __fp_parse_word_nc:N [13203](#)
- __fp_parse_word_nd:N [13203](#)
- __fp_parse_word_pc:N [13203](#)
- __fp_parse_word_pi:N [13189](#)
- __fp_parse_word_pt:N [13203](#)
- __fp_parse_word_rand:N [17346](#)
- __fp_parse_word_randint:N ... [17346](#)
- __fp_parse_word_round:N [12162](#)
- __fp_parse_word_sec:N [16234](#)
- __fp_parse_word_secd:N [16234](#)
- __fp_parse_word_sign:N [14312](#)
- __fp_parse_word_sin:N [16234](#)
- __fp_parse_word_sind:N [16234](#)
- __fp_parse_word_sp:N [13203](#)
- __fp_parse_word_sqrt:N [14312](#)
- __fp_parse_word_tan:N [16234](#)
- __fp_parse_word_tand:N [16234](#)
- __fp_parse_word_true:N [13189](#)
- __fp_parse_word_trunc:N [12156](#)
- __fp_parse_zero:
..... [636](#), [12743](#), [12763](#), [12767](#)
- __fp_pow_B:wwN [16098](#), [16133](#)
- __fp_pow_C_neg:w [16136](#), [16153](#)
- __fp_pow_C_overflow:w
..... [16141](#), [16148](#), [16169](#)
- __fp_pow_C_pack:w [16155](#), [16163](#), [16174](#)
- __fp_pow_C_pos:w [16139](#), [16158](#)
- __fp_pow_C_pos_loop:wN
..... [16159](#), [16160](#), [16167](#)
- __fp_pow_exponent:Nwnnnnnw
..... [16104](#), [16107](#), [16112](#)
- __fp_pow_exponent:wnN . [16096](#), [16101](#)
- __fp_pow_neg:www .. [744](#), [16009](#), [16176](#)
- __fp_pow_neg_aux:wNN ... [744](#), [16176](#)
- __fp_pow_neg_case:w .. [16178](#), [16199](#)
- __fp_pow_neg_case_aux:nnnnn . [16199](#)
- __fp_pow_neg_case_aux:Nnnw
..... [745](#), [16199](#)
- __fp_pow_normal_o:ww
..... [740](#), [16014](#), [16046](#)
- __fp_pow_npos_aux:NNnnw
..... [16081](#), [16085](#), [16091](#)
- __fp_pow_npos_o:Nww [741](#), [16058](#), [16075](#)
- __fp_pow_zero_or_inf:ww
..... [740](#), [16016](#), [16023](#)
- \c__fp_prec_and_int ... [12437](#), [13479](#)
- \c__fp_prec_colon_int
..... [12437](#), [13537](#), [14280](#)
- \c__fp_prec_comma_int
..... [649](#), [12437](#), [12510](#),
[13148](#), [13176](#), [13419](#), [13424](#), [13433](#)
- \c__fp_prec_comp_int
..... [12437](#), [13565](#), [13608](#)
- \c__fp_prec_end_int [652](#),
[655](#), [12437](#), [12512](#), [13255](#), [13268](#), [13402](#)
- \c__fp_prec_func_int . [649](#), [12437](#),
[13147](#), [13239](#), [13248](#), [13742](#), [13762](#)
- \c__fp_prec_hat_int ... [12437](#), [13469](#)
- \c__fp_prec_hatii_int . [12437](#), [13469](#)
- \c__fp_prec_int [11603](#), [11829](#), [11890](#),
[11917](#), [12375](#), [15844](#), [16211](#), [16214](#),
[17164](#), [17166](#), [17172](#), [17223](#), [17442](#)
- \c__fp_prec_not_int
..... [648](#), [12437](#), [13132](#), [13133](#)

\c__fp_prec_or_int [12437](#), [13481](#)
\c__fp_prec_plus_int
..... [623](#), [12437](#), [13475](#), [13477](#)
\c__fp_prec_quest_int
..... [12437](#), [13520](#), [13535](#)
\c__fp_prec_times_int
..... [12437](#), [13471](#), [13473](#)
\c__fp_prec_tuple_int
..... [649](#), [12437](#), [12511](#), [13150](#), [13178](#)
\c__fp_rand_eight_int
..... [17368](#), [17387](#), [17390](#)
\c__fp_rand_four_int
.. [17368](#), [17393](#), [17396](#), [17409](#), [17412](#)
__fp_rand_myriads:n
..... [779](#), [781](#), [17373](#), [17428](#), [17496](#)
__fp_rand_myriads_get:w [17373](#)
__fp_rand_myriads_last: [17373](#)
__fp_rand_myriads_last:w [17373](#)
__fp_rand_myriads_loop:nn ... [17373](#)
__fp_rand_o: [17417](#)
__fp_rand_o:Nw
..... [17347](#), [17354](#), [17359](#), [17417](#)
__fp_rand_o:w [17417](#)
\c__fp_rand_size_int
..... [783](#), [17368](#), [17469](#),
[17483](#), [17525](#), [17526](#), [17541](#), [17548](#)
__fp_rand_uniform: [17368](#),
[17383](#), [17405](#), [17467](#), [17486](#), [17504](#)
__fp_randint:ww [17509](#)
__fp_randint_badarg:w [17429](#)
__fp_randint_default:w [17429](#)
__fp_randint_e:wnn [17429](#)
__fp_randint_e:wwNnn [17429](#)
__fp_randint_e:wwwNnn ... [781](#), [17429](#)
__fp_randint_narrow:n ... [783](#), [17509](#)
__fp_randint_narrow:nn [17509](#)
__fp_randint_narrow:nnn [17509](#)
__fp_randint_narrow_e:nnnn .. [17429](#)
__fp_randint_o:Nw [17349](#), [17359](#), [17429](#)
__fp_randint_o:w [17429](#)
__fp_randint_wide_e:nnnn [17429](#)
__fp_randint_wide_e:wnnn [17429](#)
__fp_reverse_args:Nww [768](#),
[769](#), [11582](#), [16754](#), [16829](#), [16942](#), [17008](#)
__fp_round:NNN [613](#), [614](#), [615](#), [691](#),
[707](#), [12172](#), [12242](#), [14442](#), [14453](#),
[14697](#), [14709](#), [14851](#), [14862](#), [15046](#)
__fp_round:Nwn . [12300](#), [12353](#), [17271](#)
__fp_round:Nww . [12301](#), [12322](#), [12353](#)
__fp_round:Nwww [12302](#), [12316](#)
__fp_round_aux_o:Nw [12289](#)
__fp_round_digit:Nw .. [602](#), [616](#),
[690](#), [691](#), [707](#), [11847](#), [12256](#), [14456](#),
[14599](#), [14700](#), [14712](#), [14865](#), [15051](#)
__fp_round_name_from_cs:N
.. [12292](#), [12312](#), [12338](#), [12342](#), [12358](#)
__fp_round_neg:NNN [613](#),
[616](#), [687](#), [12267](#), [14561](#), [14576](#), [14594](#)
__fp_round_no_arg_o:Nw [12299](#), [12306](#)
__fp_round_normal:NnnwNNnn .. [12353](#)
__fp_round_normal:NNwNnn [12353](#)
__fp_round_normal:NwNNnw [12353](#)
__fp_round_normal_end:wwNnn . [12353](#)
__fp_round_o:Nw
.. [12157](#), [12159](#), [12161](#), [12165](#), [12289](#)
__fp_round_pack:Nw [12353](#)
__fp_round_return_one:
..... [614](#), [12172](#), [12178](#),
[12188](#), [12196](#), [12200](#), [12209](#), [12213](#),
[12222](#), [12229](#), [12233](#), [12271](#), [12282](#)
__fp_round_s:NNNw
.. [613](#), [615](#), [642](#), [12240](#), [12925](#), [12943](#)
__fp_round_special:NwwNnn ... [12353](#)
__fp_round_special_aux:Nw ... [12353](#)
__fp_round_to_nearest:NNN
..... [617](#), [617](#), [12165](#), [12168](#),
[12172](#), [12276](#), [12308](#), [12318](#), [17271](#)
__fp_round_to_nearest_neg:NNN [12267](#)
__fp_round_to_nearest_ninf:NNN .
..... [617](#), [12172](#), [12287](#)
__fp_round_to_nearest_ninf_-
neg:NNN [12267](#)
__fp_round_to_nearest_pinf:NNN .
..... [617](#), [12172](#), [12278](#)
__fp_round_to_nearest_pinf_-
neg:NNN [12267](#)
__fp_round_to_nearest_zero:NNN .
..... [617](#), [12172](#)
__fp_round_to_nearest_zero_-
neg:NNN [12267](#)
__fp_round_to_ninf:NNN
..... [12159](#), [12172](#), [12275](#), [12346](#)
__fp_round_to_ninf_neg:NNN .. [12267](#)
__fp_round_to_pinf:NNN
..... [12161](#), [12172](#), [12267](#), [12348](#)
__fp_round_to_pinf_neg:NNN .. [12267](#)
__fp_round_to_zero:NNN
..... [12157](#), [12172](#), [12344](#)
__fp_round_to_zero_neg:NNN .. [12267](#)
__fp_rrrot:www [11583](#), [16875](#)
__fp_sanitize:Nw
.... [682](#), [684](#), [690](#), [692](#), [701](#), [758](#),
[765](#), [11631](#), [12414](#), [12432](#), [14385](#),
[14479](#), [14651](#), [14732](#), [14880](#), [15587](#),
[15830](#), [16077](#), [16716](#), [16760](#), [16887](#)
__fp_sanitize:wN
..... [634](#), [637](#), [11631](#), [12638](#), [13139](#)
__fp_sanitize_zero:w [11631](#)

- _fp_sec_o:w [16306](#)
- _fp_set_sign_o:w
 - .. [13132](#), [14313](#), [15066](#), [15067](#), [15088](#)
- _fp_show:NN [13868](#)
- _fp_sign_aux_o:w [15055](#)
- _fp_sign_o:w [14315](#), [15055](#)
- _fp_sin_o:w [606](#), [647](#), [647](#), [767](#), [16261](#)
- _fp_sin_series_aux_o:NNwww [16668](#)
- _fp_sin_series_o:NNwww .. [747](#),
 - [759](#), [16267](#), [16282](#), [16297](#), [16312](#), [16668](#)
- _fp_small_int:wTF ... [11899](#), [12355](#)
- _fp_small_int_normal:NnwTF . [11899](#)
- _fp_small_int_test:NnnwTF . [11899](#)
- _fp_small_int_test:NnnwNw
 - [11918](#), [11921](#)
- _fp_small_int_true:wTF [11899](#)
- _fp_sqrt_auxi_o:NNNNwnnN
 - [14902](#), [14910](#)
- _fp_sqrt_auxii_o:NNnnnnnnN ...
 - [702](#), [704](#), [14912](#), [14916](#), [14996](#), [15008](#)
- _fp_sqrt_auxiii_o:wnnnnnnnn ...
 - [14913](#), [14951](#), [14997](#)
- _fp_sqrt_auxiv_o:NNNNNw [14951](#)
- _fp_sqrt_auxix_o:wnwnw [14985](#)
- _fp_sqrt_auxv_o:NNNNNw [14951](#)
- _fp_sqrt_auxvi_o:NNNNNw [14951](#)
- _fp_sqrt_auxvii_o:NNNNNw ... [14951](#)
- _fp_sqrt_auxviii_o:nnnnnnn ...
 - .. [14973](#), [14975](#), [14977](#), [14983](#), [14985](#)
- _fp_sqrt_auxx_o:NNnnnnnnn
 - [14981](#), [14999](#)
- _fp_sqrt_auxxi_o:wnnnN [14999](#)
- _fp_sqrt_auxxii_o:nnnnnnnnw ...
 - [15009](#), [15013](#)
- _fp_sqrt_auxxiii_o:w [15013](#)
- _fp_sqrt_auxxiv_o:wnnnnnnnN ...
 - [15025](#), [15028](#), [15036](#), [15038](#)
- _fp_sqrt_Newton_o:wnw
 - [702](#), [14887](#), [14898](#), [14899](#)
- _fp_sqrt_npos_auxi_o:wnnnN . [14878](#)
- _fp_sqrt_npos_auxii_o:wNNNNNNNN
 - [14878](#)
- _fp_sqrt_npos_o:w ... [14875](#), [14878](#)
- _fp_sqrt_o:w [14317](#), [14868](#)
- _fp_step:NNnnnn [14150](#)
- _fp_step:NnnnnN [14080](#)
- _fp_step:wwwN [14080](#)
- _fp_step_fp:wwwN [14080](#)
- _fp_str_if_eq_x:nn
 - .. [11949](#), [13050](#), [13064](#), [13352](#), [16049](#)
- _fp_sub_back_far_o:NnnwnnnnN ..
 - [686](#), [14488](#), [14534](#)
- _fp_sub_back_near_after:wNNNNw
 - [14494](#), [14572](#)
- _fp_sub_back_near_o:nnnnnnnnN .
 - [685](#), [14484](#), [14494](#)
- _fp_sub_back_near_pack:NNNNNNw
 - [14494](#), [14574](#)
- _fp_sub_back_not_far_o:wwwNN .
 - [14549](#), [14569](#)
- _fp_sub_back_quite_far_ii:NN [14553](#)
- _fp_sub_back_quite_far_o:wwNN .
 - [14547](#), [14553](#)
- _fp_sub_back_shift:wnnnn
 - [685](#), [14506](#), [14510](#)
- _fp_sub_back_shift_ii:ww ... [14510](#)
- _fp_sub_back_shift_iii:NNNNNNNNw
 - [14510](#)
- _fp_sub_back_shift_iv:nnnnw . [14510](#)
- _fp_sub_back_very_far_ii_-
 - o:nnNwNN [14581](#)
- _fp_sub_back_very_far_o:wwwNN
 - [14548](#), [14581](#)
- _fp_sub_eq_o:Nnnnw [14459](#)
- _fp_sub_npos_i_o:Nnnnw
 - [684](#), [14464](#), [14473](#), [14477](#)
- _fp_sub_npos_ii_o:Nnnnw [14459](#)
- _fp_sub_npos_o:NnnNw
 - [684](#), [14379](#), [14459](#)
- _fp_tan_o:w [16321](#)
- _fp_tan_series_aux_o:Nnnnw . [16722](#)
- _fp_tan_series_o:NNwww
 - [749](#), [749](#), [16328](#), [16343](#), [16722](#)
- _fp_ternary:NwN . [668](#), [13535](#), [14264](#)
- _fp_ternary_auxi:NwN
 - [668](#), [678](#), [14264](#)
- _fp_ternary_auxii:NwN
 - [668](#), [678](#), [13537](#), [14264](#)
- _fp_tmp:w [602](#),
 - [657](#), [11841](#), [11851](#), [11852](#), [11853](#),
 - [11854](#), [11855](#), [11856](#), [11857](#), [11858](#),
 - [11859](#), [11860](#), [11861](#), [11862](#), [11863](#),
 - [11864](#), [11865](#), [11866](#), [11969](#), [11971](#),
 - [12454](#), [12466](#), [12467](#), [12468](#), [12469](#),
 - [12470](#), [12471](#), [12472](#), [12531](#), [12553](#),
 - [13115](#), [13132](#), [13133](#), [13189](#), [13194](#),
 - [13195](#), [13196](#), [13197](#), [13198](#), [13199](#),
 - [13203](#), [13211](#), [13212](#), [13213](#), [13214](#),
 - [13215](#), [13216](#), [13217](#), [13218](#), [13219](#),
 - [13220](#), [13221](#), [13398](#), [13414](#), [13415](#),
 - [13438](#), [13450](#), [13468](#), [13470](#), [13472](#),
 - [13474](#), [13476](#), [13478](#), [13480](#), [13484](#),
 - [13498](#), [13499](#), [13514](#), [13515](#), [13516](#),
 - [13534](#), [13536](#), [15098](#), [15112](#), [15113](#)
- _fp_to_decimal:w
 - [17126](#), [17136](#), [17253](#), [17270](#)
- _fp_to_decimal_dispatch:w [772](#),
 - [774](#), [776](#), [14143](#), [17116](#), [17120](#), [17123](#)

__fp_to_decimal_huge:wnnnn .. [17136](#)
 __fp_to_decimal_large:Nnnw .. [17136](#)
 __fp_to_decimal_normal:wnnnnn ..
 [17136](#), [17224](#)
 __fp_to_decimal_recover:w ... [17123](#)
 __fp_to_dim:w [17238](#)
 __fp_to_dim_dispatch:w .. [775](#), [17238](#)
 __fp_to_dim_recover:w [17238](#)
 __fp_to_int:w [776](#), [17263](#), [17268](#)
 __fp_to_int_dispatch:w [17254](#)
 __fp_to_int_recover:w [17254](#)
 __fp_to_scientific:w
 [773](#), [17072](#), [17082](#)
 __fp_to_scientific_dispatch:w ..
 [771](#), [774](#), [17062](#), [17066](#), [17069](#)
 __fp_to_scientific_normal:wnnnnn
 [17082](#)
 __fp_to_scientific_normal:wNw [17082](#)
 __fp_to_scientific_recover:w . [17069](#)
 __fp_to_tl:w [17202](#), [17210](#)
 __fp_to_tl_dispatch:w
 [770](#), [774](#), [17194](#), [17198](#), [17201](#), [17339](#)
 __fp_to_tl_normal:nnnnn [17210](#)
 __fp_to_tl_recover:w [17201](#)
 __fp_to_tl_scientific:wnnnnn . [17210](#)
 __fp_to_tl_scientific:wNw ... [17210](#)
 \c__fp_trailing_shift_int
 [11786](#), [15130](#),
 [15152](#), [15225](#), [16129](#), [16614](#), [16651](#)
 __fp_trap_division_by_zero_-
 set:N [12039](#)
 __fp_trap_division_by_zero_set_-
 error: [12039](#)
 __fp_trap_division_by_zero_set_-
 flag: [12039](#)
 __fp_trap_division_by_zero_set_-
 none: [12039](#)
 __fp_trap_invalid_operation_-
 set:N [12005](#)
 __fp_trap_invalid_operation_-
 set_error: [12005](#)
 __fp_trap_invalid_operation_-
 set_flag: [12005](#)
 __fp_trap_invalid_operation_-
 set_none: [12005](#)
 __fp_trap_overflow_set:N [12065](#)
 __fp_trap_overflow_set:NnNn . [12065](#)
 __fp_trap_overflow_set_error: [12065](#)
 __fp_trap_overflow_set_flag: . [12065](#)
 __fp_trap_overflow_set_none: . [12065](#)
 __fp_trap_underflow_set:N ... [12065](#)
 __fp_trap_underflow_set_error: .
 [12065](#)
 __fp_trap_underflow_set_flag: [12065](#)
 __fp_trap_underflow_set_none: [12065](#)
 __fp_trig:NNNNNwn . [16267](#), [16282](#),
 [16297](#), [16312](#), [16327](#), [16342](#), [16359](#)
 __fp_trig_inverse_two_pi:
 [756](#), [16420](#), [16589](#)
 __fp_trig_large:ww ... [16367](#), [16584](#)
 __fp_trig_large_auxi:wwwww . [16584](#)
 __fp_trig_large_auxii:ww [16584](#)
 __fp_trig_large_auxiii:wNNNNNNNN
 [16584](#)
 __fp_trig_large_auxiv:wN [16584](#)
 __fp_trig_large_auxix:Nw [16627](#)
 __fp_trig_large_auxv:www
 [16600](#), [16607](#)
 __fp_trig_large_auxvi:wnnnnnnnn
 [16607](#)
 __fp_trig_large_auxvii:w
 [16610](#), [16627](#)
 __fp_trig_large_auxviii:w ... [16627](#)
 __fp_trig_large_auxviii:ww
 [16629](#), [16633](#)
 __fp_trig_large_auxx:wNNNNN . [16627](#)
 __fp_trig_large_auxxi:w [16627](#)
 __fp_trig_large_pack:NNNNw ...
 [16607](#), [16656](#)
 __fp_trig_small:ww
 [751](#), [757](#), [16369](#), [16373](#), [16379](#), [16666](#)
 __fp_trigd_large:ww .. [16367](#), [16381](#)
 __fp_trigd_large_auxi:nnnnwNNNN
 [16381](#)
 __fp_trigd_large_auxii:wNw .. [16381](#)
 __fp_trigd_large_auxiii:www . [16381](#)
 __fp_trigd_small:ww
 [751](#), [16369](#), [16375](#), [16418](#)
 __fp_trim_zeros:w
 [17053](#), [17177](#), [17186](#), [17237](#)
 __fp_trim_zeros_dot:w [17053](#)
 __fp_trim_zeros_end:w [17053](#)
 __fp_trim_zeros_loop:w [17053](#)
 __fp_tuple_ [14254](#), [14255](#), [14258](#), [14259](#)
 __fp_tuple_&o:ww [14237](#)
 __fp_tuple_&tuple_o:ww [14237](#)
 __fp_tuple_*o:ww [15092](#)
 __fp_tuple_+_tuple_o:ww [15098](#)
 __fp_tuple_-_tuple_o:ww [15098](#)
 __fp_tuple_/o:ww [15092](#)
 __fp_tuple_chk:w [595](#),
 [11689](#), [11695](#), [11696](#), [11773](#), [11776](#),
 [13447](#), [13657](#), [13672](#), [13697](#), [13700](#),
 [13716](#), [13717](#), [13720](#), [13978](#), [13979](#),
 [15101](#), [15102](#), [15108](#), [15109](#), [17032](#)
 __fp_tuple_compare_back:ww .. [13975](#)
 __fp_tuple_compare_back_loop:w .
 [13975](#)

- __fp_tuple_compare_back-
tuple:ww [13975](#)
 - __fp_tuple_convert:Nw
..... [17032](#), [17081](#), [17135](#), [17209](#)
 - __fp_tuple_convert_end:w [17032](#)
 - __fp_tuple_convert_loop:nNw . [17032](#)
 - __fp_tuple_count:w [11694](#)
 - __fp_tuple_count_loop:Nw [11694](#)
 - __fp_tuple_map_loop_o:nw [13697](#)
 - __fp_tuple_map_o:nw
.. [13697](#), [15085](#), [15093](#), [15095](#), [15097](#)
 - __fp_tuple_mapthread_loop_o:nw .
..... [13715](#)
 - __fp_tuple_mapthread_o:nww
..... [13715](#), [15106](#)
 - __fp_tuple_not_o:w [14228](#)
 - __fp_tuple_set_sign_aux_o:Nnw [15077](#)
 - __fp_tuple_set_sign_aux_o:w . [15077](#)
 - __fp_tuple_set_sign_o:w [15077](#)
 - __fp_tuple_to_decimal:w [17123](#)
 - __fp_tuple_to_scientific:w .. [17069](#)
 - __fp_tuple_to_tl:w [17201](#)
 - __fp_tuple_l_o:ww [14237](#)
 - __fp_tuple_l_tuple_o:ww [14237](#)
 - __fp_type_from_scan:N
.. [596](#), [11718](#), [13296](#), [13298](#), [13322](#),
[13324](#), [13335](#), [13337](#), [13939](#), [13941](#)
 - __fp_type_from_scan:w [11718](#)
 - __fp_type_from_scan_other:N ...
..... [11718](#), [11742](#), [11760](#)
 - __fp_underflow:w
.. [594](#), [608](#), [610](#), [11638](#), [12096](#), [15827](#)
 - __fp_use_i:ww
..... [718](#), [767](#), [11584](#), [15342](#), [16961](#)
 - __fp_use_i:www [11584](#)
 - __fp_use_i_until_s:nw
..... [757](#), [11579](#), [11625](#), [11891](#),
[11941](#), [16411](#), [16609](#), [16615](#), [16646](#)
 - __fp_use_ii_until_s:nnw [11579](#), [11623](#)
 - __fp_use_none_stop_f:n
..... [11576](#), [15507](#), [15508](#), [15509](#)
 - __fp_use_none_until_s:w
.. [11579](#), [14904](#), [16185](#), [16956](#), [16959](#)
 - __fp_use_s:n [11577](#)
 - __fp_use_s:nn [11577](#)
 - __fp_zero_fp:N . [11616](#), [12080](#), [12420](#)
 - __fp_l_o:ww [668](#), [14237](#)
 - __fp_l_tuple_o:ww [14237](#)
 - __fp_ [14240](#), [14247](#), [14256](#), [14257](#)
 - \futurelet [373](#)
- G**
- \gdef [374](#)
 - \GetIdInfo [6](#)
 - \gleaders [916](#)
 - \global [166](#),
[167](#), [181](#), [182](#), [183](#), [194](#), [195](#), [196](#),
[197](#), [198](#), [199](#), [200](#), [201](#), [202](#), [271](#), [375](#)
 - \globaldefs [376](#)
 - \glueexpr [631](#)
 - \glueshrink [632](#)
 - \glueshrinkorder [633](#)
 - \gluestretch [634](#)
 - \gluestretchorder [635](#)
 - \gluetomu [636](#)
 - group commands:
 - \group_align_safe_begin/end: [445](#), [804](#)
 - \group_align_safe_begin:
..... [101](#), [354](#), [358](#), [439](#), [3375](#),
[3695](#), [6313](#), [6490](#), [7693](#), [18098](#), [24495](#)
 - \group_align_safe_end:
..... [101](#), [354](#), [358](#), [3398](#), [3721](#),
[6315](#), [6490](#), [7698](#), [7704](#), [18101](#), [24516](#)
 - \group_begin: [8](#), [950](#),
[1430](#), [2434](#), [2437](#), [2440](#), [2746](#), [2948](#),
[3130](#), [3239](#), [3245](#), [3274](#), [3462](#), [3965](#),
[3992](#), [4268](#), [4291](#), [4928](#), [7253](#), [7259](#),
[7310](#), [7392](#), [7416](#), [7434](#), [7458](#), [7546](#),
[7564](#), [7772](#), [8301](#), [8322](#), [8385](#), [8667](#),
[9031](#), [9385](#), [9420](#), [9670](#), [9712](#), [10643](#),
[14237](#), [17762](#), [17799](#), [18097](#), [18104](#),
[18177](#), [18337](#), [18686](#), [18780](#), [18788](#),
[19091](#), [19591](#), [19954](#), [20047](#), [20467](#),
[20821](#), [21038](#), [21198](#), [21207](#), [21219](#),
[21228](#), [21236](#), [21354](#), [21384](#), [21879](#),
[23254](#), [23460](#), [23563](#), [24208](#), [24373](#),
[24400](#), [25045](#), [25082](#), [25096](#), [25118](#),
[25267](#), [25289](#), [25624](#), [25630](#), [26891](#)
 - \c_group_begin_token [45](#),
[118](#), [368](#), [473](#), [3826](#), [3861](#), [7416](#),
[7440](#), [18142](#), [21927](#), [21934](#), [21949](#),
[21956](#), [22039](#), [22046](#), [22061](#), [22068](#)
 - \group_end: [8](#), [8](#), [400](#),
[950](#), [1430](#), [2434](#), [2437](#), [2443](#), [2755](#),
[2951](#), [3136](#), [3241](#), [3254](#), [3332](#), [3466](#),
[3469](#), [3983](#), [4015](#), [4273](#), [4296](#), [4938](#),
[4943](#), [7261](#), [7268](#), [7395](#), [7413](#), [7433](#),
[7437](#), [7465](#), [7563](#), [7611](#), [7837](#), [8308](#),
[8333](#), [8524](#), [8712](#), [9047](#), [9389](#), [9441](#),
[9690](#), [9721](#), [10694](#), [14261](#), [17766](#),
[17807](#), [18004](#), [18102](#), [18123](#), [18184](#),
[18361](#), [18699](#), [18802](#), [19124](#), [19132](#),
[19604](#), [20012](#), [20054](#), [20061](#), [20069](#),
[20471](#), [20472](#), [20858](#), [21102](#), [21203](#),
[21214](#), [21298](#), [21360](#), [21421](#), [21885](#),
[23258](#), [23560](#), [23577](#), [24215](#), [24380](#),
[24405](#), [25073](#), [25110](#), [25117](#), [25282](#),
[25288](#), [25315](#), [25628](#), [25654](#), [26899](#)

- `\c_group_end_token` 118,
473, 7416, 7445, 18145, 21942, 22054
- `\group_insert_after:N`
. 8, 1436, 18695, 25753, 25839, 26132
- groups commands:
 - `.groups:n` 168, 11159
- H**
- `\H` 25318
- `\halign` 377
- `\hangafter` 378
- `\hangindent` 379
- `\hbadness` 380
- `\hbox` 381
- hbox commands:
 - `\hbox:n`
219, 21893, 22123, 22371, 23045, 23100
 - `\hbox_gset:Nn` 219, 21895
 - `\hbox_gset:Nw` 220, 21923
 - `\hbox_gset_end:` 220, 21923
 - `\hbox_gset_to_wd:Nnn` 219, 21909
 - `\hbox_gset_to_wd:Nnw` 220, 21945
 - `\hbox_overlap_left:n` 220, 21971
 - `\hbox_overlap_right:n` 219,
21971, 25887, 25919, 26160, 26208,
26249, 26346, 26374, 26759, 26859
 - `\hbox_set:Nn` 219, 220, 21895,
22094, 22119, 22120, 22206, 22244,
22259, 22274, 22286, 22302, 22332,
22344, 22477, 22821, 22899, 23190,
23596, 23600, 23608, 23616, 23625,
23634, 23646, 23654, 23662, 23668,
23681, 23728, 23742, 26311, 26510
 - `\hbox_set:Nw` 220, 21923, 22523
 - `\hbox_set_end:` 220, 220, 21923, 22526
 - `\hbox_set_to_wd:Nnn` . 219, 220, 21909
 - `\hbox_set_to_wd:Nnw` 220, 21945
 - `\hbox_to_wd:nn` 219, 21961, 22362
 - `\hbox_to_zero:n`
..... 219, 21961, 21972, 21974
 - `\hbox_unpack:N` 220, 21975, 22825
 - `\hbox_unpack_clear:N` 220, 21975
- hcoffin commands:
 - `\hcoffin_set:Nn`
226, 22473, 23042, 23054, 23097, 23137
 - `\hcoffin_set:Nw` 226, 22519
 - `\hcoffin_set_end:` 226, 22519
- `\hfil` 382
- `\hfill` 383
- `\hfilneg` 384
- `\hfuzz` 385
- `\hjcode` 911
- `\hoffset` 386
- `\holdinginserts` 387
- `\hpack` 912
- `\hrule` 388
- `\hsize` 389
- `\hskip` 390
- `\hss` 391
- `\ht` 392
- hundred commands:
 - `\c_one_hundred` 87, 6132
- `\hyphenation` 393
- `\hyphenationbounds` 913
- `\hyphenationmin` 914
- `\hyphenchar` 394
- `\hyphenpenalty` 395
- `\hyphenpenaltymode` 915
- I**
- `\I` 195
- `\i` 198, 25313
- `\if` 396
- if commands:
 - `\if:w` 21, 113, 311, 312,
339, 1402, 1652, 1664, 1666, 2028,
2328, 2329, 2827, 2830, 2831, 2832,
2833, 2848, 2849, 2850, 2851, 2852,
2853, 2854, 2855, 2921, 2922, 2924,
6002, 7658, 8939, 8941, 8943, 10722,
10726, 10749, 12723, 12727, 12749,
12842, 12874, 12893, 12959, 12973,
12990, 13010, 13488, 13503, 16080
 - `\if_bool:N` 99, 99, 6238, 6273
 - `\if_box_empty:N` ... 225, 21829, 21841
 - `\if_case:w` .. 89, 382, 384, 426, 603,
692, 737, 745, 2236, 4366, 4440,
5261, 5893, 5926, 7407, 9525, 11633,
11886, 11901, 12297, 12326, 13576,
13617, 14326, 14461, 14536, 14561,
14613, 15057, 15074, 15351, 15576,
15603, 15761, 15796, 15899, 15915,
15931, 15947, 15963, 15979, 16004,
16056, 16178, 16201, 16263, 16278,
16293, 16308, 16323, 16338, 16784,
16837, 16921, 16936, 16988, 17001,
17085, 17139, 17213, 17451, 18153,
18347, 18645, 18918, 19707, 19736,
19793, 20226, 20283, 20633, 20974
 - `\if_catcode:w` 21, 359, 369,
370, 482, 1402, 3510, 3817, 3859,
3871, 3888, 7440, 7445, 7450, 7455,
7462, 7469, 7474, 7479, 7484, 7489,
7494, 7504, 7531, 7730, 7735, 12476,
12683, 13000, 13047, 13350, 18142,
18145, 18299, 18301, 18303, 18305,
18307, 18309, 18311, 25636, 25637

`\if_charcode:w`
 . 21, 113, 368, 369, 387, 482, 809,
 1402, 3800, 3852, 4524, 7509, 7732,
 11889, 13905, 14267, 18212, 18236,
 18285, 18861, 18871, 19343, 20827
`\if_cs_exist:N`
 21, 1416, 2055, 2083, 2749, 7539, 7667
`\if_cs_exist:w` 21, 1416, 1444, 2064,
 2092, 2223, 6170, 6213, 6223, 24019
`\if_dim:w` 164, 9994, 10112, 10130, 10153
`\if_eof:w` 148, 9177, 9184
`\if_false:`
 . . 20, 94, 355, 358, 366, 402, 416,
 445, 802, 891, 1402, 3393, 3394,
 3482, 3486, 3768, 3773, 3784, 3872,
 3884, 3899, 3907, 4881, 4884, 5016,
 5021, 5498, 6491, 7350, 9504, 9543,
 9547, 9554, 9562, 10140, 18089,
 18131, 18180, 18183, 19100, 19119,
 19120, 19129, 19180, 19216, 19230,
 19234, 19459, 19492, 19504, 19508,
 19542, 19547, 19555, 19589, 19596,
 19601, 19649, 19871, 19888, 19892,
 21054, 21071, 21310, 21315, 21426,
 21431, 25422, 25434, 25462, 25472
`\if_hbox:N` 224, 21829, 21833
`\if_int_compare:w` 20, 88,
 416, 417, 1434, 4100, 4109, 4114,
 4350, 4405, 4406, 4412, 4424, 4440,
 5261, 5309, 5393, 5478, 5531, 5533,
 5535, 5537, 5539, 5541, 5543, 5552,
 5704, 6491, 6493, 7282, 7283, 7290,
 7291, 7292, 7293, 7298, 7299, 7339,
 7649, 9511, 10169, 11634, 11635,
 11829, 11926, 12177, 12187, 12195,
 12208, 12221, 12228, 12249, 12261,
 12270, 12281, 12382, 12387, 12458,
 12488, 12643, 12645, 12682, 12687,
 12740, 12760, 12787, 12801, 12836,
 12863, 12891, 12907, 12923, 12941,
 13000, 13020, 13036, 13049, 13063,
 13124, 13147, 13176, 13178, 13351,
 13361, 13363, 13402, 13419, 13424,
 13454, 13520, 13565, 13916, 13963,
 13966, 13997, 14006, 14009, 14014,
 14015, 14018, 14021, 14208, 14330,
 14351, 14388, 14483, 14537, 14538,
 14541, 14544, 14614, 14623, 14828,
 14901, 14954, 14958, 14962, 14980,
 15015, 15016, 15017, 15018, 15019,
 15045, 15353, 15356, 15450, 15543,
 15589, 15605, 15732, 15766, 15824,
 15833, 15873, 16049, 16051, 16062,
 16080, 16103, 16135, 16138, 16181,
 16211, 16366, 16410, 16788, 16826,
 16835, 16871, 16955, 16958, 17187,
 17387, 17393, 17409, 17441, 17579,
 17583, 17609, 17675, 17744, 17773,
 17814, 17825, 17828, 17846, 17889,
 17899, 17909, 18117, 18189, 18222,
 18230, 18253, 18277, 18328, 18343,
 18425, 18428, 18583, 18589, 18590,
 18597, 18600, 18603, 18609, 18610,
 18614, 18617, 18618, 18626, 18627,
 18628, 18634, 18664, 18665, 18915,
 18935, 18936, 18937, 18940, 18944,
 18945, 18948, 18949, 18957, 18958,
 18961, 18965, 18966, 18969, 19016,
 19038, 19050, 19059, 19067, 19070,
 19080, 19083, 19111, 19184, 19291,
 19357, 19362, 19390, 19457, 19490,
 19600, 19617, 19902, 19935, 20161,
 20244, 20273, 20338, 20351, 20362,
 20378, 20429, 20461, 20611, 20612,
 20665, 20692, 20767, 20838, 20901,
 20911, 20914, 20935, 20993, 21050,
 21067, 21090, 21239, 21268, 21308,
 21313, 21334, 21412, 21424, 21429,
 24446, 24447, 24452, 25531, 25540,
 25553, 25561, 25609, 25610, 25616
`\if_int_odd:w`
 89, 760, 5261, 5390, 5592,
 5602, 6138, 7289, 7297, 7316, 12199,
 12246, 12258, 13613, 14597, 14883,
 16222, 16636, 16675, 16685, 16728,
 16752, 16912, 18353, 18654, 19027,
 19035, 19047, 19463, 19776, 25635
`\if_meaning:w`
 20, 356, 356, 369, 676, 1402, 1744,
 1747, 1894, 1909, 1927, 1977, 1982,
 1991, 2052, 2070, 2080, 2098, 2254,
 2268, 2389, 2452, 2500, 2501, 2775,
 2798, 2807, 2992, 3004, 3005, 3420,
 3430, 3441, 3452, 3465, 3713, 3777,
 3843, 4209, 4277, 4300, 4461, 4499,
 4612, 4618, 4644, 4656, 4664, 4688,
 4917, 4933, 4948, 4956, 5280, 5330,
 5335, 5336, 5513, 6324, 6346, 6724,
 6739, 6761, 6775, 7499, 7536, 7574,
 7577, 7641, 7723, 7764, 8117, 9461,
 10083, 10146, 10659, 10683, 10700,
 10711, 11622, 11636, 11648, 11658,
 11753, 11808, 11817, 11908, 11923,
 11925, 12088, 12176, 12186, 12198,
 12211, 12212, 12231, 12232, 12246,
 12247, 12258, 12259, 12325, 12364,
 12399, 12402, 12418, 12425, 12477,
 12480, 12598, 12599, 12600, 12601,

- 12604, 12700, 12813, 12819, 13048,
 13096, 13307, 13380, 13614, 13632,
 13682, 13692, 13952, 13953, 13954,
 13955, 13956, 13957, 14189, 14201,
 14202, 14230, 14242, 14249, 14266,
 14327, 14362, 14376, 14422, 14429,
 14505, 14517, 14617, 14620, 14631,
 14684, 14757, 14827, 14830, 14837,
 14870, 14871, 14874, 15079, 15324,
 15335, 15516, 15526, 15573, 15672,
 15752, 15801, 15815, 16001, 16013,
 16026, 16029, 16032, 16035, 16061,
 16162, 16166, 16221, 16678, 16731,
 16782, 16783, 16785, 16786, 16806,
 16823, 16890, 16988, 17084, 17138,
 17212, 17283, 17288, 17440, 17686,
 17687, 18139, 18169, 18197, 18297,
 18694, 19015, 19339, 19342, 19783,
 20208, 20386, 20397, 20412, 20554,
 20587, 20947, 21190, 21267, 21320,
 21359, 25427, 25466, 25485, 25638
 \if_mode_horizontal: . 21, 1412, 6485
 \if_mode_inner: 21, 1412, 6487
 \if_mode_math: 21, 1412, 6489
 \if_mode_vertical:
 21, 1412, 6483, 23588
 \if_predicate:w 92, 94, 99, 6238, 6305
 \if_true: 20, 94, 356, 1402
 \if_vbox:N 224, 21829, 21835
 \ifabsdim 994
 \ifabsnum 995
 \ifcase 397
 \ifcat 398
 \ifcsname 637
 \ifdbbox 1195
 \ifddir 1196
 \ifdefined 158, 638
 \ifdim 399
 \ifeof 400
 \iffalse 401
 \iffontchar 639
 \ifhbox 402
 \ifhmode 403
 \ifincsname 788
 \ifinner 404
 \ifmdir 1197
 \ifmmode 405
 \ifnum 45, 60, 89, 95, 100, 164, 179, 406
 \ifodd 407
 \ifpdfabsdim 747
 \ifpdfabsnum 748
 \ifpdfprimitive 749
 \ifprimitive 874
 \iftbox 1198
 \iftdir 1199
 \iftrue 408
 \ifvbox 409
 \ifvmode 410
 \ifvoid 411
 \ifx 14, 21,
 39, 43, 49, 90, 92, 98, 123, 145, 146, 412
 \ifybox 1200
 \ifydir 1201
 \ignoreligaturesinfont 996
 \ignorespaces 413
 \IJ 25304
 \ij 25304
 image commands:
 \image_bb_restore:nTF 26282, 26505
 \image_bb_save:n 26317, 26513
 \l_image_decode_tl 26259, 26270,
 26300, 26388, 26413, 26455, 26474
 \l_image_decodearray_tl 26260,
 26296, 26301, 26414, 26452, 26456
 \image_extract_bb:n
 26383, 26390, 26862, 26863
 \l_image_interpolate_bool
 26261, 26271, 26295, 26302,
 26389, 26415, 26451, 26457, 26475
 \l_image_llx_dim 26444
 \l_image_lly_dim 26445
 \l_image_page_int
 26255, 26275, 26276, 26306, 26307,
 26381, 26411, 26412, 26438, 26439,
 26467, 26480, 26481, 26519, 26520
 \l_image_pagebox_tl
 1034, 26256, 26274, 26308,
 26309, 26382, 26409, 26410, 26440,
 26442, 26468, 26489, 26490, 26521
 \image_read_bb:n 25922, 26377
 \l_image_urx_dim 26313, 26446
 \l_image_ury_dim
 26314, 26447, 26512, 26869, 26870
 \l_image_utx_dim 26511
 \immediate 414
 in 195
 \indent 415
 inf 195
 \infty 12601, 12602
 inherit commands:
 .inherit:n 168, 11161
 \inhibitglue 1202
 \inhibitxspcode 1203
 \initcatcodetable 917
 initial commands:
 .initial:n 169, 11163
 \input 50, 159, 160, 416
 \inputlineno 417

- \insert 418
- \insertht 998
- \insertpenalties 419
- int commands:
 - \c_eight 87, [6113](#)
 - \c_eleven 87, [6113](#)
 - \c_fifteen 87, [6113](#)
 - \c_five 87, [6113](#)
 - \l_foo_int [210](#)
 - \c_four 87, [6113](#)
 - \c_fourteen 87, [6113](#)
 - \int_abs:n 76, [5271](#), 17609
 - \int_add:Nn . 78, [5433](#), 9610, 18635, 19778, 20368, 20369, 20596, 20689
 - \int_case:nn 81, [426](#), [5558](#), 5756, 5762
 - \int_case:nnn 27170
 - \int_case:nnTF 81, 5175, [5558](#), 5563, 5568, 7037, 12508, 17034, 21709, 24940, 24995, 27171
 - \int_compare:nNnTF 79, 79, 81, 81, 82, 82, 3248, 3278, 3292, 3296, 3321, 3919, 3926, 4329, 4331, 4340, 5074, 5081, 5371, 5377, [5544](#), 5582, 5638, 5646, 5655, 5661, 5688, 5691, 5752, 5840, 5846, 5852, 5872, 6026, 6045, 6047, 6089, 6391, 6564, 7076, 7078, 7083, 7092, 7112, 9254, 9340, 10349, 13769, 13981, 15104, 17017, 17162, 17164, 17483, 17517, 17524, 17525, 17526, 17554, 17572, 17620, 17622, 17662, 18472, 18672, 18684, 18847, 19153, 19155, 19948, 20557, 20785, 20800, 21005, 21284, 21728, 23490, 23696, 24119, 24202, 24431, 24658, 24660, 24663, 24806, 24828, 24862, 24865, 24899, 24902, 24909, 24922, 25037, 25580, 26275, 26306, 26411, 26438, 26480, 26519
 - \int_compare:nTF 80, 82, 82, 82, 82, 183, 550, [5491](#), 5610, 5618, 5627, 5633, 9155, 9182, 9310, 17222, 20052, 21467, 21688, 21689, 21694, 21696
 - \int_compare_p:n 80, [5491](#), 20059
 - \int_compare_p:nNn 20, 79, [5544](#), 6551, 9242, 19846, 19847, 24277, 24279, 24281, 24887, 24982, 24983, 24984, 25029, 25322, 25323, 25347, 25348
 - \int_const:Nn 77, [5365](#), 6055, 6056, 6057, 6058, 6059, 6060, 6061, 6062, 6063, 6064, 6065, 6066, 6067, 6068, 6113, 6114, 6115, 6116, 6117, 6118, 6119, 6120, 6121, 6122, 6123, 6124, 6125, 6126, 6127, 6128, 6129, 6130, 6131, 6132, 6133, 6134, 6135, 6136, 6151, 6520, 6522, 6524, 6525, 6526, 9084, 9237, 9238, 11603, 11604, 11605, 11606, 11607, 11608, 11609, 11786, 11787, 11788, 11790, 11791, 11792, 11795, 11796, 11797, 12171, 12437, 12438, 12439, 12440, 12441, 12442, 12443, 12444, 12445, 12446, 12447, 12448, 12449, 12450, 17370, 17371, 17372, 17728, 18569, 18570, 18571, 18572, 18976, 18977, 18978, 18979, 18980, 18981, 18985, 18986, 18987, 18988, 18989, 18990, 18991, 18992, 18993, 18994, 18995, 18996, 18997, 24265, 24283, 26315, 26433
 - \int_decr:N 78, [5449](#), 17834, 17835, 17836, 17887, 17888, 17897, 17898, 17907, 17908, 18132, 20635, 20995, 21068, 21269
 - \int_div_round:nn 77, [5315](#)
 - \int_div_truncate:nn . 77, 77, [783](#), [5315](#), 5767, 5865, 5885, 6523, 17541, 17547, 24456, 24463, 24465, 24474
 - \int_do_until:nn 82, [5608](#)
 - \int_do_until:nNnn 81, [5636](#)
 - \int_do_while:nn 82, [5608](#)
 - \int_do_while:nNnn 81, [5636](#)
 - \int_eval:n 13, 25, 76, 76, 76, 77, 78, 79, 80, 81, 88, 89, 239, 288, 292, 319, 429, 589, 621, 669, 693–695, 785, 786, 818, 972, 980, 2236, 2265, 2281, 3614, 3619, 3912, 3920, 3928, 4323, 4336, 4361, 4385, 4386, 4398, 4403, 4434, 4451, 4488, 5067, 5075, 5083, 5149, [5266](#), 5561, 5566, 5571, 5576, 5749, 5835, 5837, 5967, 5977, 6012, 6023, 6029, 6040, 6071, 6108, 6112, 6438, 7010, 7019, 7070, 7080, 7094, 7101, 7116, 7154, 7156, 7224, 7226, 7230, 7232, 7236, 7238, 7242, 7244, 7277, 7278, 9141, 9295, 9555, 11613, 13748, 17363, 17512, 17513, 17528, 17532, 17571, 17602, 17603, 17630, 17776, 18030, 18031, 18255, 18331, 18335, 18358, 18654, 18919, 19776, 19982, 19986, 19989, 19993, 20181, 20183, 20192, 20209, 20210, 20212, 20213, 20362, 20452, 20483, 20660, 20708, 20783, 20913, 20918, 21471, 21516, 21517, 21715, 21861, 21871, 24085, 24122, 24232, 24264, 24347, 24442, 24474, 24476, 25522, 25526, 25535,

- 25542, 25555, 25563, 25597, 25607
 \int_eval:w
 . 76, 4354, 5266, 6173, 6224, 9507,
 9516, 9540, 9552, 18082, 18317, 18327
 \int_from_alph:n 85, 6010
 \int_from_base:nn
 86, 6027, 6050, 6052, 6054
 \int_from_bin:n 85, 6049, 27173
 \int_from_binary:n 27172
 \int_from_hex:n 86, 6049, 27175
 \int_from_hexadecimal:n 27174
 \int_from_oct:n 86, 6049, 27177
 \int_from_octal:n 27176
 \int_from_roman:n 86, 6069
 \int_gadd:Nn 78, 5433
 \int_gdecr:N
 . 78, 3575, 4228, 5126, 5449, 5747,
 6971, 8162, 9224, 10327, 14173, 18397
 \int_gincr:N
 78, 3568, 4217, 5118, 5449,
 5722, 5733, 6964, 8157, 9215, 10306,
 10313, 14152, 14159, 17568, 18375,
 26432, 26779, 26907, 26963, 27008
 .int_gset:N 169, 11171
 \int_gset:Nn 78, 412, 5465, 8352
 \int_gset_eq:NN 78, 5415, 26900
 \int_gsub:Nn 78, 5433
 \int_gzero:N ... 77, 5403, 5412, 26892
 \int_gzero_new:N 77, 5409
 \int_if_even:nTF 81, 5588, 9727
 \int_if_even_p:n 81, 5588
 \int_if_exist:NTF . 78, 5410, 5412,
 5421, 6083, 6087, 19702, 19757, 26422
 \int_if_exist_p:N 78, 5421
 \int_if_odd:nTF 81, 5588, 15439
 \int_if_odd_p:n 81, 5588, 20096
 \int_incr:N 78,
 5449, 10965, 17748, 17844, 17845,
 18173, 18215, 18228, 18246, 18526,
 18527, 19594, 20017, 20201, 20294,
 20597, 20632, 20634, 20715, 20981,
 21051, 21210, 21266, 21330, 24222
 \int_log:N 87, 6109
 \int_log:n 87, 6111
 \int_max:nn 77, 777, 5271,
 15300, 16390, 20318, 21416, 21417
 \int_min:nn 77, 5271
 \int_mod:nn 77, 5315,
 5757, 5856, 5876, 6521, 17479, 24476
 \int_new:N 77, 77,
 5357, 5373, 5379, 5410, 5412, 6146,
 6147, 6148, 6149, 6495, 9362, 9365,
 9367, 9380, 10782, 17563, 17643,
 17644, 17645, 17646, 17647, 17648,
 17649, 17650, 17651, 17652, 17653,
 18068, 18069, 18070, 18071, 18552,
 18553, 18554, 18567, 18974, 18975,
 18982, 18983, 19000, 20113, 20115,
 20116, 20117, 20120, 20489, 20490,
 20491, 20492, 20493, 20494, 20495,
 20496, 20497, 20498, 20501, 20502,
 20503, 20745, 21179, 21182, 21183,
 21184, 21734, 24195, 24196, 25798,
 26393, 26829, 26910, 26911, 26989
 \int_rand:nn 86, 243,
 779, 974, 6113, 8838, 17360, 17509,
 23697, 23702, 24112, 24184, 25494
 .int_set:N 169, 11171
 \int_set:Nn 78, 288, 2441, 3249, 3306,
 5465, 9200, 9202, 9346, 9348, 9363,
 9373, 9386, 9422, 9428, 9438, 10970,
 17656, 17658, 17660, 17683, 17684,
 17699, 17707, 17708, 17720, 17721,
 17732, 17733, 17734, 17750, 17753,
 18127, 18190, 18514, 19784, 20114,
 20176, 20178, 20259, 20314, 20315,
 20328, 20339, 20363, 20381, 20430,
 20522, 20547, 20568, 20664, 20699,
 21215, 21363, 21389, 21870, 21872,
 21880, 21881, 21882, 21883, 24223
 \int_set_eq:NN 78, 5415,
 17700, 17741, 18625, 19081, 19085,
 19094, 19096, 19139, 19192, 19494,
 19593, 19606, 19704, 20136, 20153,
 20199, 20200, 20250, 20360, 20361,
 20413, 20468, 20524, 20527, 20544,
 20546, 20549, 20563, 20566, 20598,
 20599, 20603, 20739, 21321, 26888
 \int_show:N 86, 6105
 \int_show:n 86, 431, 972, 6107
 \int_step... 212
 \int_step_function:nN 83, 5664, 24213
 \int_step_function:nnN
 83, 5664, 7391,
 7396, 7399, 9251, 17803, 20604, 21280
 \int_step_function:nnnN . 83, 242,
 242, 422, 673, 5664, 5746, 21392, 21400
 \int_step_inline:nn . 83, 5716, 17574
 \int_step_inline:nnn
 . 83, 5716, 9093, 9260, 20538, 21739
 \int_step_inline:nnnn . 83, 674, 5716
 \int_step_variable:nN 83, 5716
 \int_step_variable:nnn 83
 \int_step_variable:nnnN 83
 \int_step_variable:nnnn 83
 \int_step_variable:nnnnN ... 83, 5716
 \int_step_variable:nnnnn 83

\int_sub:Nn 78, 5433, 9618,
 18629, 19298, 20416, 20424, 20433
 \int_to_Alph:n 84, 85, 5770
 \int_to_alph:n 84, 84, 85, 5770
 \int_to_arabic:n 84, 5749
 \int_to_Base:n 85
 \int_to_base:n 85
 \int_to_Base:nn . . . 85, 86, 5834, 5961
 \int_to_base:nn
 85, 86, 5834, 5957, 5959, 5963
 \int_to_bin:n . . . 85, 85, 85, 5956, 27179
 \int_to_binary:n 27178
 \int_to_Hex:n 85, 86, 5956, 18850
 \int_to_hex:n 85, 86, 5956, 27181
 \int_to_hexadecimal:n 27180
 \int_to_oct:n 85, 86, 5956, 27183
 \int_to_octal:n 27182
 \int_to_Roman:n 85, 86, 5964
 \int_to_roman:n 85, 86, 5964
 \int_to_symbols:nnn
 84, 84, 5750, 5772, 5804
 \int_until_do:nn 82, 5608
 \int_until_do:nNnn 82, 5636
 \int_use:N
 76, 79, 615, 620, 3570, 3572, 4219,
 4223, 5119, 5125, 5473, 5725, 5736,
 6966, 6968, 8156, 8164, 8268, 8796,
 9203, 9217, 9342, 10309, 10316,
 10971, 14155, 14162, 18377, 19113,
 19186, 19257, 19268, 19278, 19282,
 19294, 19295, 19301, 19302, 19308,
 19309, 19477, 20096, 20169, 20171,
 20189, 20190, 20191, 20292, 20305,
 20306, 20650, 20700, 20730, 20840,
 20852, 20949, 21215, 21729, 24234,
 26276, 26307, 26322, 26412, 26425,
 26437, 26439, 26520, 26781, 26812,
 26966, 26972, 26979, 27011, 27019
 \int_value:w . . . 88, 416, 440, 549,
 593, 599, 603, 616, 622, 629, 632,
 637, 638, 644, 670, 671, 680, 688,
 696, 759, 773, 785, 786, 802, 974,
 1697, 2033, 4323, 4324, 4336, 4354,
 4361, 4384, 4385, 4386, 4398, 4434,
 5261, 5269, 5270, 5275, 5276, 5288,
 5289, 5290, 5301, 5302, 5303, 5322,
 5324, 5325, 5342, 5350, 5351, 5352,
 5494, 5498, 5528, 5682, 5683, 5684,
 5710, 5920, 5953, 6156, 6173, 6224,
 6234, 6337, 6340, 6438, 7277, 7278,
 9507, 9516, 10121, 10363, 11680,
 11681, 11682, 11683, 11684, 11698,
 11846, 11907, 11925, 12245, 12367,
 12381, 12383, 12385, 12388, 12424,
 12563, 12593, 12594, 12631, 12639,
 12770, 12775, 12777, 12786, 12790,
 12827, 12835, 12838, 12844, 12855,
 12866, 12872, 12873, 12876, 12919,
 12929, 12931, 12947, 12949, 12972,
 12986, 13064, 13066, 13140, 13228,
 13759, 13951, 13984, 14337, 14338,
 14339, 14341, 14387, 14390, 14393,
 14416, 14418, 14439, 14441, 14450,
 14452, 14456, 14474, 14481, 14487,
 14497, 14499, 14513, 14521, 14529,
 14573, 14575, 14591, 14593, 14596,
 14599, 14653, 14661, 14663, 14665,
 14667, 14670, 14673, 14675, 14694,
 14696, 14700, 14706, 14708, 14712,
 14734, 14737, 14745, 14747, 14750,
 14751, 14752, 14753, 14768, 14771,
 14774, 14777, 14786, 14789, 14792,
 14795, 14802, 14804, 14810, 14818,
 14820, 14822, 14848, 14850, 14859,
 14861, 14865, 14882, 14903, 14907,
 14919, 14922, 14925, 14928, 14931,
 14934, 14937, 14940, 14944, 14956,
 14960, 14964, 14967, 14988, 14990,
 14992, 15002, 15026, 15029, 15041,
 15043, 15049, 15052, 15073, 15123,
 15128, 15130, 15137, 15140, 15143,
 15146, 15149, 15152, 15161, 15173,
 15181, 15183, 15193, 15195, 15202,
 15211, 15213, 15216, 15219, 15222,
 15225, 15238, 15240, 15248, 15250,
 15258, 15260, 15270, 15273, 15276,
 15283, 15298, 15316, 15319, 15375,
 15389, 15391, 15397, 15410, 15412,
 15414, 15438, 15454, 15461, 15462,
 15506, 15508, 15509, 15510, 15551,
 15553, 15588, 15595, 15602, 15623,
 15625, 15627, 15629, 15642, 15646,
 15647, 15648, 15649, 15650, 15655,
 15660, 15662, 15668, 15685, 15686,
 15687, 15688, 15689, 15690, 15695,
 15697, 15699, 15701, 15703, 15708,
 15710, 15712, 15714, 15716, 15718,
 15740, 15748, 15764, 15769, 15773,
 15832, 15881, 15892, 16079, 16115,
 16117, 16120, 16123, 16126, 16129,
 16136, 16139, 16141, 16145, 16167,
 16169, 16197, 16364, 16396, 16405,
 16587, 16588, 16589, 16611, 16621,
 16630, 16648, 16657, 16664, 16674,
 16718, 16727, 16762, 16805, 16822,
 16878, 16889, 16900, 17110, 17186,
 17233, 17279, 17287, 17289, 17291,
 17383, 17389, 17395, 17405, 17411,

- 17467, 17479, 17486, 17504, 17512,
 17513, 17602, 17603, 17620, 17622,
 17625, 17630, 17776, 18082, 18094,
 18273, 18315, 18317, 18327, 18335,
 18354, 18356, 18364, 18843, 19327,
 19333, 19365, 19367, 19376, 19377,
 19501, 19924, 19939, 20761, 20762,
 20773, 21304, 21335, 21337, 24112,
 24122, 25535, 25542, 25555, 25563
 \int_while_do:nn 82, [5608](#)
 \int_while_do:nNnn 82, [5636](#)
 \int_zero:N 77, 77, [5403](#),
 5410, 9473, 10962, 18128, 18129,
 18130, 18229, 19093, 19296, 19741,
 20049, 20135, 20521, 20545, 20823,
 21209, 24210, 26255, 26381, 26467
 \int_zero_new:N 77, [5409](#)
 \c_max_int . 87, [199](#), [822](#), [874](#), [6135](#),
 18600, 18614, 20599, 21858, 21864
 \c_nine 87, [6113](#)
 \c_one
 87, [433](#), 5451, 5454, 5457, 5460, [6113](#)
 \c_seven 87, [6113](#)
 \c_six 87, [6113](#)
 \c_sixteen 87, [6113](#)
 \c_ten 87, [6113](#)
 \c_thirteen 87, [6113](#)
 \c_three 87, [6113](#)
 \g_tmpa_int 88, [6146](#)
 \l_tmpa_int 2, 87, [205](#), [6146](#)
 \g_tmpb_int 88, [6146](#)
 \l_tmpb_int 2, 87, [6146](#)
 \c_twelve 87, [6113](#)
 \c_two 87, [6113](#)
 \c_zero
 87, [295](#), [312](#), [312](#), [980](#), [1453](#), 2031,
 2033, 5371, 5404, 5406, 5478, 5486,
 5688, 5691, [6113](#), 6491, 6493, 10169
 int internal commands:
 __int_abs:N [5271](#)
 __int_case:nnTF [5558](#)
 __int_case:nw [5558](#)
 __int_case_end:nw [5558](#)
 __int_compare:nnN [417](#), [5491](#)
 __int_compare:NNw ... [416](#), [417](#), [5491](#)
 __int_compare:Nw [416](#), [417](#), [5491](#)
 __int_compare:w [416](#), [5491](#)
 __int_compare_!=:NNw [5491](#)
 __int_compare_<:NNw [5491](#)
 __int_compare_<=:NNw [5491](#)
 __int_compare_=:NNw [5491](#)
 __int_compare_==:NNw [5491](#)
 __int_compare_>:NNw [5491](#)
 __int_compare_>=:NNw [5491](#)
 __int_compare_end=:NNw ... [417](#), [5491](#)
 __int_compare_error:
 [415](#), [416](#), [5476](#), 5494, 5496
 __int_compare_error:Nw
 [415](#), [417](#), [417](#), [5476](#), 5516
 __int_constdef:Nw [5365](#)
 __int_div_truncate:NwNw [5315](#)
 __int_eval:w . 288, [410](#), [411](#), [416](#),
 5261, 5267, 5269, 5270, 5272, 5276,
 5283, 5284, 5289, 5290, 5296, 5297,
 5302, 5303, 5317, 5318, 5322, 5324,
 5325, 5342, 5345, 5346, 5350, 5351,
 5352, 5368, 5387, 5430, 5435, 5438,
 5441, 5444, 5467, 5470, 5494, 5528,
 5546, 5548, 5552, 5589, 5592, 5599,
 5602, 5667, 5671, 5675, 5682, 5683,
 5684, 5710, 5893, 5920, 5926, 5953
 __int_eval_end: . 5261, 5269, 5276,
 5326, 5342, 5353, 5387, 5435, 5438,
 5441, 5444, 5467, 5470, 5547, 5552,
 5592, 5602, 5893, 5920, 5926, 5953
 __int_from_alph:N [428](#), [6010](#)
 __int_from_alph:nN [428](#), [6010](#)
 __int_from_base:N [429](#), [6027](#)
 __int_from_base:nnN [429](#), [6027](#)
 __int_from_roman:NN [6069](#)
 \c__int_from_roman_C_int [6055](#)
 \c__int_from_roman_c_int [6055](#)
 \c__int_from_roman_D_int [6055](#)
 \c__int_from_roman_d_int [6055](#)
 __int_from_roman_error:w [6069](#)
 \c__int_from_roman_I_int [6055](#)
 \c__int_from_roman_i_int [6055](#)
 \c__int_from_roman_L_int [6055](#)
 \c__int_from_roman_l_int [6055](#)
 \c__int_from_roman_M_int [6055](#)
 \c__int_from_roman_m_int [6055](#)
 \c__int_from_roman_V_int [6055](#)
 \c__int_from_roman_v_int [6055](#)
 \c__int_from_roman_X_int [6055](#)
 \c__int_from_roman_x_int [6055](#)
 \c__int_max_constdef_int [5365](#)
 __int_maxmin:wwN [5271](#)
 \c__int_minus_one
 6150, 6151, 6152, 6155
 __int_mod:ww [5315](#)
 __int_pass_signs:wn
 [428](#), [6000](#), 6014, 6031
 __int_pass_signs_end:wn [6000](#)
 __int_show:nN [6105](#)
 __int_step:NNnnnn [5716](#)
 __int_step:NwnnN [5664](#)
 __int_step:wwN [5664](#)

- __int_tmp:w [414](#), [414](#), [5425](#),
[5433](#), [5436](#), [5439](#), [5442](#), [5465](#), [5468](#)
- __int_to_Base:nn [5834](#)
- __int_to_base:nn [5834](#)
- __int_to_Base:nnN [5834](#)
- __int_to_base:nnN [5834](#)
- __int_to_Base:nnnN [5834](#)
- __int_to_base:nnnN [5834](#)
- __int_to_Letter:n [5834](#)
- __int_to_letter:n [5834](#)
- __int_to_roman:N [5964](#)
- __int_to_roman:w
[416](#), [427](#), [1434](#), [5261](#), [5504](#), [5967](#), [5977](#)
- __int_to_Roman_aux:N [5976](#), [5979](#), [5982](#)
- __int_to_Roman_c:w [5964](#)
- __int_to_roman_c:w [5964](#)
- __int_to_Roman_d:w [5964](#)
- __int_to_roman_d:w [5964](#)
- __int_to_Roman_i:w [5964](#)
- __int_to_roman_i:w [5964](#)
- __int_to_Roman_l:w [5964](#)
- __int_to_roman_l:w [5964](#)
- __int_to_Roman_m:w [5964](#)
- __int_to_roman_m:w [5964](#)
- __int_to_Roman_Q:w [5964](#)
- __int_to_roman_Q:w [5964](#)
- __int_to_Roman_v:w [5964](#)
- __int_to_roman_v:w [5964](#)
- __int_to_Roman_x:w [5964](#)
- __int_to_roman_x:w [5964](#)
- __int_to_symbols:nnnn [5750](#)
- __int_value:w [6156](#)
- intarray commands:
 - \intarray_count:N
... [199](#), [199](#), [199](#), [17576](#), [17583](#), [17594](#)
 - \intarray_gset:Nnn .. [199](#), [785](#), [17596](#)
 - \intarray_item:Nn ... [199](#), [785](#), [17624](#)
 - \intarray_new:Nn [199](#),
[784](#), [17564](#), [18564](#), [18565](#), [18566](#),
[20504](#), [20505](#), [21185](#), [21186](#), [21187](#)
- intarray internal commands:
 - __intarray_bounds:NNnTF
..... [17577](#), [17607](#), [17634](#)
 - __intarray_bounds_error:NNn . [17577](#)
 - \g_intarray_font_int
..... [17563](#), [17568](#), [17570](#)
 - __intarray_gset:Nnn [17596](#)
 - __intarray_gset:Nww .. [17600](#), [17605](#)
 - __intarray_gset_overflow:Nnn . [17596](#)
 - __intarray_gset_overflow:NNnn ..
..... [17610](#), [17616](#)
 - __intarray_item:Nn [17624](#)
 - __intarray_item:Nw ... [17628](#), [17632](#)
- \c__intarray_sp_dim
..... [17562](#), [17570](#), [17597](#)
- \interactionmode [640](#)
- \interlinepenalties [641](#)
- \interlinepenalty [420](#)
- ior commands:
 - \ior_close:N .. [141](#), [142](#), [142](#), [9132](#),
[9153](#), [9767](#), [23484](#), [23521](#), [23559](#), [23576](#)
 - \ior_get:NN
..... [143](#), [143](#), [144](#), [148](#), [9194](#), [9210](#)
 - \ior_get_str:NN [27184](#)
 - \ior_if_eof:N [523](#)
 - \ior_if_eof:NFT
... [145](#), [9178](#), [9222](#), [9229](#), [9762](#), [9776](#)
 - \ior_if_eof_p:N [145](#), [9178](#)
 - \ior_list_streams: [9983](#)
 - \ior_log_list: . [142](#), [9165](#), [9985](#), [9986](#)
 - \ior_log_streams: [9983](#)
 - \ior_map_break: [144](#), [9205](#), [9223](#), [23554](#)
 - \ior_map_break:n [145](#), [9205](#)
 - \ior_map_inline:Nn .. [144](#), [9209](#), [23479](#)
 - \ior_new:N
.. [141](#), [9107](#), [9109](#), [9110](#), [9653](#), [23457](#)
 - \ior_open:Nn [141](#),
[540](#), [9111](#), [23462](#), [23485](#), [23522](#), [23575](#)
 - \ior_open:NnTF [142](#), [9112](#), [9115](#)
 - \ior_show_list: [142](#), [9165](#), [9983](#), [9984](#)
 - \ior_str_get:NN [143](#), [9196](#), [9212](#), [27185](#)
 - \ior_str_map_inline:Nn
..... [144](#), [9209](#), [23513](#), [23545](#)
 - \c_term_ior
... [148](#), [9084](#), [9107](#), [9155](#), [9161](#), [9182](#)
 - \g_tmpa_ior [148](#), [9109](#)
 - \g_tmpb_ior [148](#), [9109](#)
- ior internal commands:
 - \l_ior_file_name_str
..... [9114](#), [9117](#), [9118](#), [9121](#)
 - \l_ior_internal_tl [9209](#)
 - __ior_list:N [9165](#)
 - __ior_map_inline:NNn [9209](#)
 - __ior_map_inline:NNNn [9209](#)
 - __ior_map_inline_loop:NNN ... [9209](#)
 - __ior_new:N [521](#), [9126](#), [9140](#)
 - __ior_open_stream:Nn [9130](#)
 - \l_ior_stream_tl
..... [9090](#), [9133](#), [9141](#), [9149](#)
 - \g_ior_streams_prop
..... [9091](#), [9150](#), [9158](#), [9172](#)
 - \g_ior_streams_seq
..... [9085](#), [9133](#), [9159](#), [9160](#)
- iow commands:
 - \iow_char:N . [146](#), [1670](#), [8804](#), [8806](#),
[8807](#), [8909](#), [8930](#), [8931](#), [9361](#), [9715](#),
[15998](#), [18017](#), [18020](#), [18021](#), [18046](#),

- 18047, 18054, 18055, 18829, 18831, 18833, 18835, 18837, 18839, 19433, 19434, 19973, 20086, 20087, 20088, 20109, 21439, 21442, 21443, 21448, 21482, 21491, 21495, 21500, 21520, 21522, 21523, 21524, 21527, 21529, 21536, 21540, 21543, 21544, 21547, 21549, 21553, 21555, 21561, 21563, 21567, 21569, 21573, 21578, 21580, 21622, 21624, 21629, 21631, 21637, 21642, 21647, 21651, 21661, 21664, 21668, 21669, 21673, 21681, 21744
- \iow_close:N 142, 142, 9286, 9308
- \iow_indent:n 8852, 9400, 9431, 9435, 12120, 12132
- \l_iow_line_count_int 147, 147, 531, 815, 9362, 9439, 9475, 18474, 18478
- \iow_list_streams: 9983
- \iow_log:n 145, 289, 302, 1712, 2141, 3957, 8341, 8342, 8343, 8487, 9356
- \iow_log_list: . 142, 9320, 9989, 9990
- \iow_log_streams: 9983
- \iow_new:N 141, 9274, 9276, 9277
- \iow_newline: 146, 146, 146, 147, 289, 372, 500, 528, 8297, 8313, 8315, 9360, 9429, 9436, 9901, 18424, 20020, 23218, 23219, 23220, 24058, 24060, 24063, 24070
- \iow_now:Nn 145, 145, 145, 146, 146, 9350, 9356, 9357, 9358, 9359, 24294
- \iow_open:Nn 142, 9283
- \iow_shipout:Nn 146, 146, 146, 528, 9335, 24307
- \iow_shipout_x:Nn 146, 146, 146, 528, 9332
- \iow_show_list: 142, 9320, 9987, 9988
- \iow_term:n 145, 2141, 8311, 8347, 8348, 8349, 8513, 9356, 21730
- \iow_wrap:nnnN 139, 140, 140, 146, 146, 147, 147, 147, 372, 531, 972, 3942, 3957, 8289, 8290, 8342, 8348, 8485, 8492, 9403, 9415, 9418
- \c_log_iow 148, 524, 9237, 9310, 9356, 9357
- \c_term_iow .. 148, 524, 9237, 9251, 9254, 9274, 9310, 9316, 9358, 9359
- \g_tmpa_iow 148, 9276
- \g_tmpb_iow 148, 9276
- iow internal commands:
 - \l_iow_file_name_str 9282, 9285, 9288, 9296
 - _iow_indent:n 530, 9400, 9431
 - _iow_indent_error:n 530, 9400, 9435
 - \l_iow_indent_int 9379, 9473, 9491, 9602, 9610, 9618
 - \l_iow_indent_tl 9379, 9474, 9490, 9601, 9611, 9619, 9620
 - \l_iow_line_break_bool 9383, 9469, 9596, 9609, 9617, 9625, 9627, 9632, 9634
 - \l_iow_line_part_tl 533, 534, 9381, 9471, 9483, 9504, 9561, 9564, 9595, 9608, 9616, 9639
 - \l_iow_line_target_int 536, 9365, 9438, 9597, 9602, 9628
 - \l_iow_line_tl 9381, 9470, 9487, 9576, 9592, 9608, 9616, 9638, 9639, 9644, 9646
 - _iow_list:N 9320
 - _iow_new:N 9278, 9294
 - \l_iow_newline_tl 9364, 9436, 9437, 9439, 9643
 - \l_iow_one_indent_int 9366, 9610, 9618
 - \l_iow_one_indent_tl 529, 9366, 9611
 - _iow_open_stream:Nn 9283
 - _iow_set_indent:n 529, 9366
 - \l_iow_stream_tl 9257, 9287, 9295, 9303
 - \g_iow_streams_prop 9258, 9304, 9313, 9327
 - \g_iow_streams_seq 9246, 9287, 9314, 9315
 - _iow_tmp:w 534, 9477, 9501, 9557, 9589
 - _iow_unindent:w ... 529, 9366, 9620
 - _iow_with:nNnn 9338
 - _iow_wrap_break:w 9543, 9557
 - _iow_wrap_break_end:w ... 534, 9557
 - _iow_wrap_break_first:w 9557
 - _iow_wrap_break_loop:w 9557
 - _iow_wrap_break_none:w 9557
 - _iow_wrap_chunk:nw 9475, 9477, 9612, 9621, 9628
 - _iow_wrap_do: 9440, 9444
 - _iow_wrap_end: 9623
 - _iow_wrap_end:n 9630
 - _iow_wrap_end_chunk:w 532, 9495, 9502, 9593
 - \c_iow_wrap_end_marker_tl 9385, 9449
 - _iow_wrap_fix_newline:w 9444
 - _iow_wrap_indent: 9606
 - _iow_wrap_indent:n 9606
 - \c_iow_wrap_indent_marker_tl ... 9385, 9408
 - _iow_wrap_line:nw 532, 535, 9489, 9493, 9502, 9600

- __iow_wrap_line_aux:Nw [9502](#)
 - __iow_wrap_line_end:NnnnnnnN [9502](#)
 - __iow_wrap_line_end:nw
..... [534](#), [9502](#), [9577](#), [9578](#), [9587](#)
 - __iow_wrap_line_loop:w [9502](#)
 - \c__iow_wrap_marker_tl
..... [529](#), [532](#), [9385](#), [9501](#)
 - __iow_wrap_newline: [9623](#)
 - __iow_wrap_newline:n [9623](#)
 - \c__iow_wrap_newline_marker_tl ..
..... [531](#), [9385](#), [9464](#)
 - __iow_wrap_next:nw . [9477](#), [9555](#), [9597](#)
 - __iow_wrap_next_line:w .. [9549](#), [9590](#)
 - __iow_wrap_start:w [9444](#)
 - __iow_wrap_store_do:n
..... [9548](#), [9626](#), [9633](#), [9636](#)
 - \l__iow_wrap_tl ... [531](#), [531](#), [536](#),
[536](#), [9384](#), [9434](#), [9442](#), [9446](#), [9448](#),
[9451](#), [9453](#), [9456](#), [9472](#), [9640](#), [9642](#)
 - __iow_wrap_trim:N
..... [536](#), [9578](#), [9626](#), [9633](#), [9648](#)
 - __iow_wrap_trim:w [9648](#)
 - __iow_wrap_unindent: [9606](#)
 - __iow_wrap_unindent:n [9614](#)
 - \c__iow_wrap_unindent_marker_tl .
..... [9385](#), [9410](#)
- J**
- \J [197](#)
 - \j [25314](#)
 - \jcharwidowpenalty [1204](#)
 - \jfam [1205](#)
 - \jfont [1206](#)
 - \jis [1207](#)
 - job commands:
 - \c_job_name_tl [27166](#)
 - \jobname [421](#)
- K**
- \k [25318](#)
 - \kanjiskip [1208](#)
 - \kansuji [1209](#)
 - \kansujichar [1210](#)
 - \kcatcode [1211](#)
 - \kchar [1230](#)
 - \kchardef [1231](#)
 - \kern [422](#)
 - kernel internal commands:
 - __kernel_chk_cs_exist:N ... [288](#),
[299](#), [1589](#), [2007](#), [2011](#), [2015](#), [2019](#), [2757](#)
 - __kernel_chk_defined:NTF .. [288](#),
[465](#), [496](#), [2405](#), [2424](#), [3938](#), [5246](#),
[6185](#), [6292](#), [7131](#), [8178](#), [13874](#), [21139](#)
 - __kernel_chk_expr:nNnN [288](#),
[301](#), [1676](#), [5267](#), [5272](#), [5283](#), [5284](#),
[5296](#), [5297](#), [5317](#), [5318](#), [5345](#), [5346](#),
[5368](#), [5430](#), [5546](#), [5548](#), [5589](#), [5599](#),
[5667](#), [5671](#), [5675](#), [10034](#), [10076](#),
[10086](#), [10087](#), [10099](#), [10100](#), [10124](#),
[10126](#), [10262](#), [10266](#), [10270](#), [10330](#),
[10337](#), [10358](#), [10423](#), [10465](#), [10479](#),
[10489](#), [10499](#), [10562](#), [10601](#), [21756](#)
 - __kernel_chk_if_free_cs:N
..... [472](#), [498](#), [2145](#), [2165](#), [2213](#),
[3058](#), [3065](#), [3071](#), [4601](#), [4727](#), [5360](#),
[5383](#), [7417](#), [7419](#), [7429](#), [7881](#), [10000](#),
[10390](#), [10524](#), [17567](#), [21765](#), [24078](#)
 - __kernel_chk_var_exist:N
.. [288](#), [288](#), [299](#), [434](#), [1589](#), [3093](#),
[3099](#), [3111](#), [3112](#), [3119](#), [3120](#), [6166](#)
 - __kernel_chk_var_global:N . [288](#),
[300](#), [1589](#), [3098](#), [3149](#), [3152](#), [3155](#),
[3176](#), [3179](#), [3182](#), [3185](#), [3208](#), [3211](#),
[3214](#), [3217](#), [5405](#), [5418](#), [5439](#), [5442](#),
[5455](#), [5458](#), [5468](#), [6249](#), [6252](#), [6266](#),
[10014](#), [10040](#), [10049](#), [10056](#), [10067](#),
[10404](#), [10429](#), [10441](#), [10449](#), [10539](#),
[10571](#), [10579](#), [10585](#), [10593](#), [21785](#),
[21793](#), [21847](#), [21901](#), [21915](#), [21930](#),
[21952](#), [21999](#), [22013](#), [22027](#), [22042](#),
[22064](#), [25380](#), [25408](#), [25414](#), [25444](#)
 - __kernel_chk_var_local:N
..... [288](#), [300](#),
[1589](#), [3092](#), [3140](#), [3143](#), [3146](#), [3164](#),
[3167](#), [3170](#), [3173](#), [3196](#), [3199](#), [3202](#),
[3205](#), [5403](#), [5415](#), [5433](#), [5436](#), [5449](#),
[5452](#), [5465](#), [6243](#), [6246](#), [6263](#), [10012](#),
[10037](#), [10045](#), [10053](#), [10064](#), [10402](#),
[10426](#), [10438](#), [10446](#), [10536](#), [10568](#),
[10576](#), [10582](#), [10590](#), [21782](#), [21790](#),
[21844](#), [21895](#), [21909](#), [21923](#), [21945](#),
[21993](#), [22007](#), [22021](#), [22035](#), [22057](#),
[22077](#), [25377](#), [25396](#), [25402](#), [25440](#)
 - __kernel_chk_var_scope:NN
..... [288](#), [288](#), [299](#),
[1589](#), [3062](#), [3068](#), [4598](#), [4697](#), [5366](#),
[10005](#), [10395](#), [10529](#), [17564](#), [24075](#)
 - __kernel_cs_parm_from_arg_-
count:nnTF .. [288](#), [1882](#), [2231](#), [2278](#)
 - __kernel_debug_log:n
[289](#), [302](#), [1707](#), [2147](#), [2934](#), [8200](#), [10979](#)
 - __kernel_deprecation_code:nn ...
..... [1723](#), [6153](#), [9953](#), [27234](#)
 - __kernel_deprecation_error:Nnn .
..... [1763](#), [1784](#), [6154](#), [9955](#), [27150](#)
 - __kernel_exp_not:w [289](#),
[335](#), [2731](#), [2733](#), [2738](#), [2743](#), [3741](#),

- 3767, 3791, 5509, 7360, 24323, 24487
- _kernel_expl_bool 234, 237, 252, 266, 1401
- _kernel_file_input_pop: . 289, 9791
- _kernel_file_input_push:n 289, 9791
- _kernel_file_missing:n 289, 9112, 9785, 9795, 24371, 24398
- _kernel_file_name_sanitiz:nN . 289, 9285, 9710, 9741, 9787, 9963, 9971
- _kernel_if_debug:TF 289, 1529, 1535, 1566, 1589, 1646, 1676, 1707, 1723, 1740, 1795, 1814, 3088, 6192, 8925, 21720, 21735
- _kernel_intarray_gset:Nnn 785, 17596, 20541, 20625, 20627, 20629, 20655, 20658, 20713, 21242, 21244, 21250, 21258, 21260, 21263, 21324, 21326, 21328, 21341, 21347, 21351
- _kernel_intarray_item:Nn . 786, 17624, 20620, 20641, 20644, 20666, 20693, 20754, 20755, 20778, 20779, 20787, 20793, 20795, 20802, 20808, 20810, 20867, 20871, 21247, 21374
- _kernel_ior_open:Nn 289, 9121, 9130, 9760, 9775
- _kernel_iow_with:Nnn 289, 372, 500, 528, 3946, 3948, 8318, 8320, 8515, 8517, 9338, 9352
- _kernel_msg_error:nn 290, 2122, 8705, 10740, 10759, 17883, 19073, 19106, 19154, 19157, 19619, 19875, 21003, 21091, 22715, 24240
- _kernel_msg_error:nnnn 290, 1542, 1550, 1557, 1562, 1598, 1607, 1750, 1895, 1937, 1978, 1983, 2122, 2315, 2322, 2410, 2799, 2993, 3349, 4032, 4702, 8528, 8705, 9730, 9788, 10840, 10899, 10915, 11064, 11082, 11587, 12000, 18011, 19112, 19316, 19717, 19730, 19769, 19893, 20839, 20846, 21105, 21888, 22447, 23318, 23989, 24204
- _kernel_msg_error:nnnnn 290, 1886, 1915, 1997, 2122, 2154, 2280, 2883, 3013, 3036, 8207, 8554, 8705, 10824, 10879, 10937, 10951, 11073, 11448, 11996, 17876, 19293, 19358, 19581, 21007, 21027, 22594
- _kernel_msg_error:nnnnnn 290, 1669, 8705, 9402, 17607, 18028, 21291, 21414, 27157
- _kernel_msg_error:nnnnnnn 290, 2898, 8705, 17618, 27237
- _kernel_msg_expandable_-error:nn 291, 2752, 4718, 6469, 7284, 7286, 7294, 7300, 7340, 7876, 9048, 12519, 17356, 17362, 18824
- _kernel_msg_expandable_-error:nnn 291, 2515, 3607, 5189, 5487, 5693, 6198, 7052, 7955, 9048, 10288, 12526, 12542, 12547, 12614, 12671, 12710, 12716, 13053, 13058, 13069, 13076, 13167, 13181, 13381, 13407, 14122, 18910, 23309
- _kernel_msg_expandable_-error:nnnn 291, 1700, 9048, 13539, 13560, 14283, 17519, 18849
- _kernel_msg_expandable_-error:nnnnn 291, 9048, 9414, 12111, 13772, 17422, 17634, 27154
- _kernel_msg_expandable_-error:nnnnnn 291, 9048
- _kernel_msg_fatal:nn 290, 8705, 9136, 9290
- _kernel_msg_fatal:nnn 290, 8705
- _kernel_msg_fatal:nnnn 290, 8705
- _kernel_msg_fatal:nnnnn 290, 8705
- _kernel_msg_fatal:nnnnnn 290, 8705
- _kernel_msg_info:nn 291, 8710
- _kernel_msg_info:nnn 291, 8710
- _kernel_msg_info:nnnn 291, 8710
- _kernel_msg_info:nnnnn 291, 8710
- _kernel_msg_info:nnnnnn 291, 8710
- _kernel_msg_new:nn 290, 8659, 8755, 8757, 8759, 8761, 8763, 8781, 8837, 8913, 8934, 8983, 8985, 8987, 8989, 8991, 8993, 8995, 8997, 8999, 9006, 9013, 9020, 9919, 11563, 12139, 12141, 12143, 12145, 12147, 12149, 12151, 13794, 13796, 13798, 13800, 13802, 13804, 13806, 13808, 13810, 13812, 13814, 13816, 13818, 13820, 13825, 14175, 14177, 14179, 17352, 18488, 21438, 21440, 21445, 21698, 23246
- _kernel_msg_new:nnnn 290, 8659, 8713, 8721, 8729, 8736, 8747, 8765, 8774, 8786, 8793, 8800, 8809, 8818, 8824, 8830, 8839, 8846, 8855, 8861, 8868, 8875, 8882, 8889, 8897, 8905, 8927, 8935, 8948, 8961, 8973, 9913, 9925, 9932, 9939, 9944, 10768, 11530, 11533, 11539, 11545, 11551, 11557, 11973, 12113, 12128, 18016, 18034, 18041, 18050, 21451, 21458, 21464, 21474, 21480, 21504, 21511, 21519, 21526, 21533, 21539, 21546,

- 21552, 21560, 21566, 21572, 21582,
21589, 21598, 21601, 21609, 21615,
21621, 21628, 21635, 21645, 21656,
21666, 21676, 21685, 21691, 23230,
23237, 23240, 23323, 23995, 24229
- _kernel_msg_set:nnn 290, 8659
- _kernel_msg_set:nnnn 290, 8659
- _kernel_msg_warning:nn
. 290, 8710, 19609
- _kernel_msg_warning:nnn
. 290, 8710, 19526,
19530, 19571, 19633, 19671, 19689
- _kernel_msg_warning:nnnn
. 290, 8710, 19221, 19372
- _kernel_msg_warning:nnnnn
. 290, 1772, 8710
- _kernel_msg_warning:nnnnnn
. 290, 2918, 8642, 8710, 27241
- _kernel_patch:nnNNpn
. 291, 291, 292, 292, 304, 305,
347, 1795, 2006, 2010, 2014, 2018,
2145, 2757, 2930, 3062, 3068, 3109,
3117, 3140, 3143, 3146, 3149, 3152,
3155, 3164, 3167, 3170, 3173, 3176,
3179, 3182, 3185, 3196, 3199, 3202,
3205, 3208, 3211, 3214, 3217, 4598,
4697, 5403, 5405, 5415, 5418, 5449,
5452, 5455, 5458, 6165, 6219, 6243,
6246, 6249, 6252, 6263, 6266, 8198,
10005, 10012, 10014, 10045, 10049,
10395, 10402, 10404, 10529, 10536,
10539, 10576, 10579, 10976, 17564,
18777, 20126, 20145, 20184, 20268,
20321, 20513, 20532, 20647, 20816,
21782, 21785, 21790, 21793, 21844,
21847, 21895, 21901, 21909, 21915,
21923, 21930, 21945, 21952, 21993,
21999, 22007, 22013, 22021, 22027,
22035, 22042, 22057, 22064, 22077,
24016, 24075, 25377, 25380, 25396,
25402, 25408, 25414, 25440, 25444
- _kernel_patch_args:nnnNNpn 292,
1810, 5365, 5427, 10031, 10420, 10556
- _kernel_patch_args:nnNpn
. 292, 292,
1810, 5266, 5271, 5281, 5294, 5315,
5343, 5664, 10075, 10084, 10097,
10259, 10329, 10336, 10355, 10478,
10486, 10496, 10598, 17273, 21753
- _kernel_patch_conditional:nnNpn
. 291, 304, 1795, 6210
- _kernel_patch_conditional_
args:nnnNNpn 292, 1810
- _kernel_patch_conditional_
args:nnNpn 292,
1810, 5544, 5588, 5598, 10122, 10462
- _kernel_patch_deprecation:nnNpn
. 303, 303, 1740,
6496, 6498, 6500, 6503, 6505, 7869,
9959, 9967, 9974, 9981, 9983, 9985,
9987, 9989, 18058, 18060, 18495,
18499, 22379, 22383, 22387, 22390
- \g_kernel_prg_map_int
. 292, 361, 422,
554, 813, 1401, 3568, 3570, 3572,
3575, 4217, 4219, 4223, 4228, 5118,
5119, 5125, 5126, 5722, 5725, 5733,
5736, 5747, 6495, 6964, 6966, 6968,
6971, 8156, 8157, 8162, 8164, 9215,
9217, 9224, 10306, 10309, 10313,
10316, 10327, 14152, 14155, 14159,
14162, 14173, 18375, 18377, 18397
- _kernel_primitive:NN
. 265, 274, 283, 284, 285,
286, 287, 288, 289, 290, 291, 292,
293, 294, 295, 296, 297, 298, 299,
300, 301, 302, 303, 304, 305, 306,
307, 308, 309, 310, 311, 312, 313,
314, 315, 316, 317, 318, 319, 320,
321, 322, 323, 324, 325, 326, 327,
328, 329, 330, 331, 332, 333, 334,
335, 336, 337, 338, 339, 340, 341,
342, 343, 344, 345, 346, 347, 348,
349, 350, 351, 352, 353, 354, 355,
356, 357, 358, 359, 360, 361, 362,
363, 364, 365, 366, 367, 368, 369,
370, 371, 372, 373, 374, 375, 376,
377, 378, 379, 380, 381, 382, 383,
384, 385, 386, 387, 388, 389, 390,
391, 392, 393, 394, 395, 396, 397,
398, 399, 400, 401, 402, 403, 404,
405, 406, 407, 408, 409, 410, 411,
412, 413, 414, 415, 416, 417, 418,
419, 420, 421, 422, 423, 424, 425,
426, 427, 428, 429, 430, 431, 432,
433, 434, 435, 436, 437, 438, 439,
440, 441, 442, 443, 444, 445, 446,
447, 448, 449, 450, 451, 452, 453,
454, 455, 456, 457, 458, 459, 460,
461, 462, 463, 464, 465, 466, 467,
468, 469, 470, 471, 472, 473, 474,
475, 476, 477, 478, 479, 480, 481,
482, 483, 484, 485, 486, 487, 488,
489, 490, 491, 492, 493, 494, 495,
496, 497, 498, 499, 500, 501, 502,
503, 504, 505, 506, 507, 508, 509,
510, 511, 512, 513, 514, 515, 516,

517, 518, 519, 520, 521, 522, 523,
 524, 525, 526, 527, 528, 529, 530,
 531, 532, 533, 534, 535, 536, 537,
 538, 539, 540, 541, 542, 543, 544,
 545, 546, 547, 548, 549, 550, 551,
 552, 553, 554, 555, 556, 557, 558,
 559, 560, 561, 562, 563, 564, 565,
 566, 567, 568, 569, 570, 571, 572,
 573, 574, 575, 576, 577, 578, 579,
 580, 581, 582, 583, 584, 585, 586,
 587, 588, 589, 590, 591, 592, 593,
 594, 595, 596, 597, 598, 599, 600,
 601, 602, 603, 604, 605, 606, 607,
 608, 609, 610, 611, 612, 613, 614,
 615, 616, 617, 618, 619, 621, 622,
 623, 624, 625, 626, 627, 628, 629,
 630, 631, 632, 633, 634, 635, 636,
 637, 638, 639, 640, 641, 642, 643,
 644, 645, 646, 647, 648, 649, 650,
 651, 652, 653, 654, 655, 656, 657,
 658, 659, 660, 661, 662, 663, 664,
 665, 666, 667, 668, 669, 670, 671,
 672, 673, 674, 675, 676, 677, 678,
 679, 680, 681, 682, 683, 684, 685,
 686, 687, 688, 689, 690, 691, 692,
 693, 694, 695, 696, 697, 698, 700,
 702, 703, 704, 705, 706, 707, 708,
 710, 711, 712, 713, 714, 715, 716,
 718, 719, 720, 721, 722, 723, 724,
 725, 726, 727, 728, 729, 730, 731,
 732, 733, 734, 736, 737, 738, 739,
 740, 741, 742, 743, 744, 745, 746,
 747, 748, 749, 750, 751, 752, 753,
 754, 755, 756, 757, 758, 759, 760,
 761, 762, 763, 764, 765, 766, 767,
 768, 769, 770, 771, 772, 773, 774,
 775, 776, 777, 778, 779, 780, 781,
 782, 783, 784, 785, 786, 787, 788,
 789, 790, 791, 792, 793, 794, 795,
 796, 801, 813, 815, 816, 817, 818,
 819, 820, 821, 822, 823, 824, 825,
 827, 829, 831, 832, 833, 835, 836,
 837, 838, 839, 840, 842, 844, 845,
 847, 849, 850, 851, 852, 853, 854,
 855, 856, 857, 858, 859, 860, 861,
 862, 863, 864, 865, 866, 867, 868,
 869, 870, 871, 872, 873, 874, 875,
 876, 877, 878, 879, 880, 881, 883,
 885, 887, 888, 889, 890, 891, 893,
 895, 896, 897, 898, 899, 900, 901,
 902, 903, 905, 906, 908, 909, 910,
 911, 912, 913, 914, 915, 916, 917,
 918, 919, 920, 921, 922, 923, 924,
 925, 926, 927, 929, 930, 931, 932,
 933, 934, 935, 936, 937, 938, 939,
 940, 941, 942, 943, 944, 945, 946,
 947, 948, 949, 950, 951, 952, 953,
 954, 955, 957, 958, 959, 960, 961,
 962, 963, 964, 966, 967, 969, 970,
 972, 973, 974, 975, 976, 977, 978,
 979, 980, 981, 983, 984, 985, 986,
 987, 988, 989, 990, 991, 992, 993,
 994, 995, 996, 998, 999, 1001, 1003,
 1005, 1006, 1007, 1008, 1009, 1010,
 1011, 1012, 1013, 1014, 1015, 1016,
 1017, 1018, 1019, 1020, 1021, 1022,
 1023, 1024, 1025, 1026, 1027, 1028,
 1029, 1030, 1031, 1032, 1033, 1034,
 1035, 1036, 1037, 1038, 1039, 1040,
 1041, 1042, 1043, 1044, 1045, 1046,
 1047, 1048, 1050, 1052, 1053, 1054,
 1055, 1057, 1058, 1059, 1060, 1062,
 1063, 1065, 1067, 1068, 1069, 1070,
 1071, 1073, 1075, 1076, 1077, 1078,
 1080, 1081, 1082, 1083, 1084, 1085,
 1086, 1087, 1088, 1089, 1090, 1091,
 1092, 1093, 1094, 1095, 1096, 1097,
 1098, 1099, 1100, 1101, 1102, 1103,
 1104, 1105, 1106, 1107, 1108, 1109,
 1110, 1111, 1112, 1113, 1114, 1115,
 1116, 1117, 1119, 1121, 1122, 1124,
 1126, 1127, 1128, 1129, 1130, 1131,
 1132, 1134, 1136, 1138, 1139, 1140,
 1141, 1142, 1143, 1144, 1145, 1146,
 1147, 1148, 1149, 1151, 1153, 1154,
 1155, 1156, 1157, 1158, 1159, 1160,
 1161, 1162, 1163, 1164, 1165, 1166,
 1167, 1168, 1169, 1171, 1173, 1174,
 1175, 1176, 1177, 1178, 1179, 1180,
 1181, 1182, 1183, 1184, 1185, 1186,
 1187, 1188, 1189, 1190, 1191, 1192,
 1193, 1194, 1195, 1196, 1197, 1198,
 1199, 1200, 1201, 1202, 1203, 1204,
 1205, 1206, 1207, 1208, 1209, 1210,
 1211, 1212, 1213, 1214, 1215, 1216,
 1217, 1218, 1219, 1220, 1221, 1222,
 1223, 1224, 1225, 1226, 1227, 1228,
 1229, 1230, 1231, 1232, 1233, 1234
 __kernel_register_log:N
 292, 2414, 6109, 10376,
 10377, 10511, 10512, 10613, 10614
 __kernel_register_show:N
 292, 292,
 371, 2414, 6105, 10372, 10507, 10609
 __kernel_register_show_aux:NN 2414
 __kernel_register_show_aux:nNN 2414
 __kernel_show:NN 2432
 __kernel_str_to_other:n ... 292,

- [292, 378, 381, 385, 4263, 4315, 4376](#)
- `__kernel_str_to_other_fast:n`
 [292, 4224, 4244, 4286, 9372, 9448, 18795, 19912](#)
- `__kernel_str_to_other_fast-loop:w` [4286](#)
- `__kernel_tl_to_str:w`
 . [293, 359, 1427, 3442, 3511, 3601, 3778, 4095, 4199, 4689, 10728, 10751](#)
- keys commands:
- `\l_keys_choice_int`
 [167, 169, 171, 171, 172, 10782, 10962, 10965, 10970, 10971](#)
- `\l_keys_choice_tl`
 [167, 169, 171, 172, 10782, 10969](#)
- `\keys_define:nn` [166, 8850, 10797](#)
- `\keys_if_choice_exist:nnnTF`
 [175, 11499](#)
- `\keys_if_choice_exist_p:nnn`
 [175, 11499](#)
- `\keys_if_exist:nnTF`
 [175, 588, 11492, 11516](#)
- `\keys_if_exist_p:nn` [175, 11492](#)
- `\l_keys_key_tl` . [173, 10785, 10900, 10916, 11345, 11431, 11433, 11467](#)
- `\keys_log:nn` [175, 11507](#)
- `\l_keys_path_tl` [173, 10789, 10825, 10845, 10854, 10863, 10867, 10881, 10893, 10895, 10897, 10909, 10911, 10913, 10928, 10933, 10938, 10946, 10948, 10949, 10952, 10967, 10990, 10995, 11004, 11008, 11015, 11019, 11023, 11030, 11037, 11048, 11054, 11058, 11074, 11083, 11091, 11126, 11327, 11334, 11357, 11360, 11399, 11403, 11411, 11413, 11414, 11425, 11428, 11449, 11474, 11475](#)
- `\keys_set:nn` [165, 169, 173, 173, 174, 11025, 11030, 11223, 11247, 11255, 11303, 11312](#)
- `\keys_set_filter:nnn` [175, 11258](#)
- `\keys_set_filter:nnnN` [175, 11258](#)
- `\keys_set_groups:nnn` [175, 11258](#)
- `\keys_set_known:nn` [174, 11233](#)
- `\keys_set_known:nnN` . [174, 582, 11233](#)
- `\keys_show:nn` [175, 175, 11507](#)
- `\l_keys_value_tl` [173, 10795, 11074, 11402, 11405, 11407, 11415, 11435, 11445, 11459, 11469](#)
- keys internal commands:
- `__keys_bool_set:Nn`
 [10889, 11100, 11102, 11104, 11106](#)
- `__keys_bool_set_inverse:Nn`
 [10905, 11108, 11110, 11112, 11114](#)
- `__keys_check_groups:` [11361, 11369](#)
- `__keys_choice_find:n` [10922, 11472](#)
- `__keys_choice_make:`
 [10892, 10908, 10921, 10956, 11116](#)
- `__keys_choice_make:N` [10921](#)
- `__keys_choice_make_aux:N` [10921](#)
- `__keys_choices_make:nn`
 [10955, 11118, 11120, 11122, 11124](#)
- `__keys_choices_make:Nnn` [10955](#)
- `__keys_cmd_set:nn`
 [10893, 10895, 10897, 10909, 10911, 10913, 10948, 10949, 10966, 10976, 11023, 11030, 11091, 11126](#)
- `\c__keys_code_root_tl`
 [10775, 10978, 10983, 11019, 11411, 11414, 11431, 11433, 11442, 11444, 11456, 11458, 11495, 11503, 11522](#)
- `\c__keys_default_root_tl`
 [10775, 10990, 10995, 11399, 11403](#)
- `__keys_default_set:n` [10902, 10918, 10985, 11136, 11138, 11140, 11142](#)
- `__keys_define:n` [10802, 10806](#)
- `__keys_define:nn` [10802, 10806](#)
- `__keys_define:nnn` [10797](#)
- `__keys_define_aux:nn` [10806](#)
- `__keys_define_code:n` . [10820, 10871](#)
- `__keys_define_code:w` [10871](#)
- `__keys_execute:`
 [11338, 11365, 11387, 11391, 11409](#)
- `__keys_execute:nn` [11409, 11474, 11475](#)
- `__keys_execute_unknown:` [11409](#)
- `\l__keys_filtered_bool`
 [10791, 11271, 11279, 11281, 11285, 11293, 11295, 11364, 11385, 11390](#)
- `__keys_find_key_module:w` [11315](#)
- `\l__keys_groups_clist` [10784, 11001, 11002, 11009, 11359, 11374](#)
- `\c__keys_groups_root_tl`
 [10775, 11004, 11008, 11357, 11360](#)
- `__keys_groups_set:n` [10999, 11160](#)
- `__keys_inherit:n` [11012, 11162](#)
- `\c__keys_inherit_root_tl`
 [10775, 11015, 11425, 11428](#)
- `__keys_initialise:n`
 [11017, 11164, 11166, 11168, 11170](#)
- `__keys_keys_set_known:nn` [11233](#)
- `__keys_meta_make:n` [11021, 11180](#)
- `__keys_meta_make:nn` [11021, 11182](#)
- `\l__keys_module_tl`
 [10786, 10798, 10801, 10803, 10847, 10848, 10854, 11026, 11224, 11227, 11229, 11318, 11323, 11333, 11339, 11347, 11349, 11442, 11444, 11449](#)

- __keys_multichoice_find:n [10924](#), [11472](#)
- __keys_multichoice_make: [10921](#), [10958](#), [11184](#)
- __keys_multichoices_make:nn ... [10955](#), [11186](#), [11188](#), [11190](#), [11192](#)
- \l_keys_no_value_bool [10787](#), [10808](#), [10813](#), [10873](#), [11071](#), [11080](#), [11317](#), [11322](#), [11397](#), [11468](#)
- \l_keys_only_known_bool [10788](#), [11246](#), [11254](#), [11256](#), [11421](#)
- __keys_parent:n [10928](#), [10933](#), [10938](#), [11425](#), [11428](#), [11479](#)
- __keys_parent:w [11479](#)
- __keys_property_find:n [10818](#), [10829](#)
- __keys_property_find:w [10829](#)
- __keys_property_search:w [10855](#), [10859](#), [10868](#)
- \l_keys_property_tl [10790](#), [10819](#), [10822](#), [10825](#), [10831](#), [10832](#), [10839](#), [10851](#), [10864](#), [10876](#), [10877](#), [10880](#), [10884](#)
- \c_keys_props_root_tl ... [10781](#), [10819](#), [10877](#), [10884](#), [11099](#), [11101](#), [11103](#), [11105](#), [11107](#), [11109](#), [11111](#), [11113](#), [11115](#), [11117](#), [11119](#), [11121](#), [11123](#), [11125](#), [11127](#), [11129](#), [11131](#), [11133](#), [11135](#), [11137](#), [11139](#), [11141](#), [11143](#), [11145](#), [11147](#), [11149](#), [11151](#), [11153](#), [11155](#), [11157](#), [11159](#), [11161](#), [11163](#), [11165](#), [11167](#), [11169](#), [11171](#), [11173](#), [11175](#), [11177](#), [11179](#), [11181](#), [11183](#), [11185](#), [11187](#), [11189](#), [11191](#), [11193](#), [11195](#), [11197](#), [11199](#), [11201](#), [11203](#), [11205](#), [11207](#), [11209](#), [11211](#), [11213](#), [11215](#), [11217](#), [11219](#), [11221](#)
- __keys_remove_spaces:n .. [10801](#), [10831](#), [10967](#), [11227](#), [11331](#), [11474](#), [11490](#), [11495](#), [11503](#), [11514](#), [11523](#)
- \l_keys_selective_bool [10791](#), [11302](#), [11311](#), [11313](#), [11336](#)
- \l_keys_selective_seq [10793](#), [11298](#), [11301](#), [11306](#), [11372](#)
- __keys_set:n [11228](#), [11315](#)
- __keys_set:nn [11228](#), [11315](#)
- __keys_set:nnn [11223](#)
- __keys_set_aux: [11315](#)
- __keys_set_aux:nnn [11315](#)
- __keys_set_filter:nnn [11258](#)
- __keys_set_filter:nnnnN [11258](#)
- __keys_set_groups:nnn [11258](#)
- __keys_set_known:nn .. [11248](#), [11252](#)
- __keys_set_known:nnnN [11233](#)
- __keys_set_selective: [11315](#)
- __keys_set_selective:nn [11258](#)
- __keys_set_selective:nnn [11258](#)
- __keys_set_selective:nnnn [11258](#)
- __keys_show:Nnn [11507](#)
- __keys_store_unused: [11366](#), [11386](#), [11392](#), [11409](#)
- \l_keys_tmp_bool [10796](#), [11371](#), [11378](#), [11383](#)
- \c_keys_type_root_tl [10775](#), [10928](#), [10933](#), [10946](#)
- __keys_undefine: [11014](#), [11031](#), [11218](#)
- \l_keys_unused_clist [582](#), [10794](#), [11234](#), [11238](#), [11240](#), [11241](#), [11259](#), [11263](#), [11265](#), [11266](#), [11465](#)
- __keys_validate_cleanup:w ... [11041](#)
- __keys_validate_forbidden: .. [11041](#)
- __keys_validate_required: ... [11041](#)
- \c_keys_validate_root_tl [10775](#), [11048](#), [11054](#), [11058](#), [11413](#)
- __keys_value_or_default:n [11335](#), [11395](#)
- __keys_value_requirement:nn ... [11041](#), [11220](#), [11222](#)
- __keys_variable_set:NnnN [11088](#), [11128](#), [11130](#), [11132](#), [11134](#), [11144](#), [11146](#), [11148](#), [11150](#), [11152](#), [11154](#), [11156](#), [11158](#), [11172](#), [11174](#), [11176](#), [11178](#), [11194](#), [11196](#), [11198](#), [11200](#), [11202](#), [11204](#), [11206](#), [11208](#), [11210](#), [11212](#), [11214](#), [11216](#)
- keyval commands:
 - \keyval_parse:NNn [177](#), [10629](#), [10802](#), [11228](#)
- keyval internal commands:
 - __keyval_action: [10708](#)
 - __keyval_def:Nn . [10710](#), [10730](#), [10760](#)
 - __keyval_def_aux:n [10760](#)
 - __keyval_def_aux:w [10760](#)
 - __keyval_empty_key: .. [10754](#), [10758](#)
 - \l_keyval_key_tl [10626](#), [10710](#), [10711](#), [10724](#), [10734](#)
 - __keyval_loop:NNw [10632](#), [10638](#), [10698](#)
 - __keyval_sanitise_aux:w [10642](#)
 - __keyval_sanitise_comma: [10637](#), [10642](#)
 - __keyval_sanitise_comma_auxi:w . [10642](#)
 - __keyval_sanitise_comma_auxii:w [10642](#)
 - __keyval_sanitise_equals: [10636](#), [10642](#)
 - __keyval_sanitise_equals_auxi:w [10642](#)

- `__keyval_sanitise_equals_-auxii:w` [10642](#)
- `\l__keyval_sanitise_tl`
 [10628](#), [10635](#), [10639](#), [10648](#),
 [10650](#), [10654](#), [10661](#), [10663](#), [10672](#),
 [10674](#), [10678](#), [10685](#), [10687](#), [10696](#)
- `__keyval_split:NNw` ... [10703](#), [10708](#)
- `__keyval_split_tidy:w` [10708](#)
- `__keyval_split_value:NNw` [10708](#)
- `\l__keyval_value_tl`
 [10626](#), [10730](#), [10735](#)
- `\kuten` [1212](#), [1232](#)
- L**
- `\L` [25305](#)
- `\l` [25305](#)
- `l3kernel` [232](#), [23331](#)
- `l3kernel.charcat` [232](#), [23360](#)
- `l3kernel.filemdfivesum` [232](#), [23366](#)
- `l3kernel.filemoddate` [232](#), [23378](#)
- `l3kernel.filesize` [232](#), [23423](#)
- `l3kernel.strcmp` [232](#), [23433](#)
- `\label` [25366](#)
- `\language` [423](#)
- `\lastallocatedtoks` [17714](#)
- `\lastbox` [424](#)
- `\lastkern` [425](#)
- `\lastlinefit` [642](#)
- `\lastnamedcs` [918](#)
- `\lastnodetype` [643](#)
- `\lastpenalty` [426](#)
- `\lastsavedboxresourceindex` [999](#)
- `\lastsavedimageresourceindex` [1001](#)
- `\lastsavedimageresourcepages` [1003](#)
- `\lastskip` [427](#)
- `\lastxpos` [1005](#)
- `\lastypos` [1006](#)
- `\latelua` [919](#)
- LaTeX3 error commands:
 - `\LaTeX3_error:` [517](#), [518](#)
- `\lccode` [166](#), [181](#), [194](#), [196](#), [198](#), [200](#), [202](#), [428](#)
- `\leaders` [429](#)
- `\left` [430](#)
- left commands:
 - `\c_left_brace_str` [59](#), [4576](#), [18871](#),
 [19244](#), [19248](#), [19268](#), [19282](#), [19307](#),
 [19788](#), [19869](#), [20927](#), [20962](#), [20987](#)
- `\leftghost` [978](#)
- `\lefthyphenmin` [431](#)
- `\leftmargin` [789](#)
- `\leftmargin` [789](#)
- `\leftskip` [432](#)
- `\leqno` [433](#)
- `\let` [1](#), [40](#), [271](#), [272](#), [434](#)
- `\letcharcode` [920](#)
- `\letterspacefont` [790](#)
- `\limits` [435](#)
- `\LineBreak` [80](#), [81](#), [82](#), [83](#), [84](#),
 [85](#), [86](#), [87](#), [105](#), [112](#), [113](#), [114](#), [122](#), [124](#)
- `\linedir` [979](#)
- `\linepenalty` [436](#)
- `\lineskip` [437](#)
- `\lineskiplimit` [438](#)
- `\linewidth` [22496](#), [22542](#)
- `\ln` [16104](#), [16107](#)
- `ln` [191](#)
- `\localbrokenpenalty` [980](#)
- `\localinterlinepenalty` [981](#)
- `\lcalleftbox` [983](#)
- `\lcalrightbox` [984](#)
- `\loccount` [9100](#), [9267](#)
- `\loctoks` [17686](#), [17687](#), [17713](#)
- `\long` [274](#), [439](#), [7601](#), [7605](#)
- `\LongText` [76](#), [110](#), [134](#)
- `\looseness` [440](#)
- `\lower` [441](#)
- `\lowercase` [442](#)
- `\lpcode` [791](#)
- lua commands:
 - `\lua_escape:n` [232](#), [23291](#)
 - `\lua_escape_x:n` . [232](#), [4079](#), [4094](#),
 [23289](#), [23291](#), [23969](#), [24289](#), [24302](#)
 - `\lua_now:n` [231](#), [23291](#)
 - `\lua_now_x:n` [231](#), [4080](#),
 [4083](#), [7325](#), [23290](#), [23291](#), [23966](#), [24288](#)
 - `\lua_shipout:n` [231](#), [23291](#)
 - `\lua_shipout_x:n` ... [231](#), [23291](#), [24301](#)
- luaescapestring [921](#)
- luafunction [922](#)
- luatex commands:
 - `\luatex_alignmark:D` [877](#), [1288](#)
 - `\luatex_aligntab:D` [878](#), [1289](#)
 - `\luatex_attribute:D` [879](#), [1290](#)
 - `\luatex_attributedef:D` [880](#), [1291](#)
 - `\luatex_automaticdiscretionary:D` [882](#)
 - `\luatex_automatichyphenmode:D` .. [884](#)
 - `\luatex_automatichyphenpenalty:D` [886](#)
 - `\luatex_begincsname:D` [887](#)
 - `\luatex_bodydir:D` [976](#), [1327](#), [1391](#)
 - `\luatex_boxdir:D` [977](#), [1328](#)
 - `\luatex_breakafterdirmode:D` ... [888](#)
 - `\luatex_catcodetable:D` [889](#), [1292](#)
 - `\luatex_clearmarks:D` [890](#), [1293](#)
 - `\luatex_crampeddisplaystyle:D` ...
 [892](#), [1294](#)
 - `\luatex_crampedscriptscriptstyle:D`
 [894](#), [1295](#)
 - `\luatex_crampedscriptstyle:D` ...
 [895](#), [1297](#)

- \luatex_crampedtextstyle:D 896, 1298
- \luatex_directlua:D 897,
1281, 1282, 4080, 11954, 23287, 24269
- \luatex_dviextension:D 898
- \luatex_dvifedback:D 899
- \luatex_dvivariable:D 900
- \luatex_etoksapp:D 901
- \luatex_etokspre:D 902
- \luatex_expanded:D ... 905, 1384, 1496
- \luatex_explicitdiscretionary:D 907
- \luatex_explicithyphenpenalty:D 904
- \luatex_firstvalidlanguage:D .. 908
- \luatex_fontid:D 909, 1299
- \luatex_formatname:D 910, 1300
- \luatex_gleaders:D 916, 1301
- \luatex_hjcode:D 911
- \luatex_hpack:D 912
- \luatex_hyphenationbounds:D ... 913
- \luatex_hyphenationmin:D 914
- \luatex_hyphenpenaltymode:D ... 915
- \luatex_if_engine:TF
..... 27188, 27190, 27192
- \luatex_if_engine_p: 27186
- \luatex_initcatcodetable:D 917, 1302
- \luatex_lastnamedcs:D 918
- \luatex_latelua:D ... 919, 1303, 23288
- \luatex_leftghost:D 978, 1329
- \luatex_letcharcode:D 920
- \luatex_linedir:D 979
- \luatex_localbrokenpenalty:D ...
..... 980, 1330
- \luatex_localinterlinepenalty:D .
..... 982, 1331
- \luatex_localleftbox:D 983, 1333
- \luatex_localrightbox:D ... 984, 1334
- \luatex_luaescapestring:D .. 376,
921, 1304, 4079, 11958, 11959, 23286
- \luatex_luafunction:D 922, 1305
- \luatex luatexbanner:D 923
- \luatex luatexrevision:D 924
- \luatex luatexversion:D
..... 925, 1347, 1366,
1454, 4077, 5391, 6139, 6545, 9242
- \luatex_mathdelimitersmode:D .. 926
- \luatex_mathdir:D 985, 1335
- \luatex_mathdisplayskipmode:D .. 928
- \luatex_matheqnogapstep:D 929
- \luatex_mathnolimitsmode:D 930
- \luatex_mathoption:D 931
- \luatex_mathpenaltiesmode:D ... 932
- \luatex_mathrulesfam:D 933
- \luatex_mathscriptboxmode:D ... 935
- \luatex_mathscriptsmode:D 934
- \luatex_mathstyle:D 936, 1306
- \luatex_mathsurroundmode:D 937
- \luatex_mathsurroundskip:D 938
- \luatex_nohrule:D 939
- \luatex_nokerns:D 940, 1307
- \luatex_noligs:D 941, 1308
- \luatex_nospaces:D 942
- \luatex_novrule:D 943
- \luatex_outputbox:D 944, 1309
- \luatex_pagebottomoffset:D 945, 1336
- \luatex_pagedir:D 986, 1337, 1392
- \luatex_pageleftoffset:D .. 946, 1310
- \luatex_pagerightoffset:D . 947, 1339
- \luatex_pagetopoffset:D ... 948, 1311
- \luatex_pardir:D 987, 1341
- \luatex_pdfextension:D
..... 949, 25835, 25836,
25844, 25845, 26171, 26172, 26179,
26180, 26185, 26186, 26191, 26192
- \luatex_pdffeedback:D 950
- \luatex_pdfvariable:D 951
- \luatex_postexhyphenchar:D 952, 1312
- \luatex_posthyphenchar:D .. 953, 1313
- \luatex_prebinoppenalty:D 954
- \luatex_predisplaygapfactor:D .. 956
- \luatex_preexhyphenchar:D . 957, 1314
- \luatex_prehyphenchar:D ... 958, 1315
- \luatex_prerelpenalty:D 959
- \luatex_rightghost:D 988, 1342
- \luatex_savecatcodetable:D 960, 1316
- \luatex_scantextokens:D ... 961, 1317
- \luatex_setfontid:D 962
- \luatex_shapemode:D 963
- \luatex_suppressifcsnameerror:D .
..... 965, 1318
- \luatex_suppresslongerror:D 966, 1320
- \luatex_suppressmathparerror:D ..
..... 968, 1321
- \luatex_suppressoutererror:D ...
..... 969, 1323
- \luatex_suppressprimitiveerror:D 971
- \luatex_textdir:D 989, 1343
- \luatex_toksapp:D 972
- \luatex_tokspre:D 973
- \luatex_tpack:D 974
- \luatex_vpack:D 975
- luatex internal commands:
 - _luatex_escape_x:n .. 23286, 23296
 - _luatex_now_x:n 23286, 23291
 - _luatex_shiphout_x:n 23293
 - _luatex_shipout_x:n 23286
- \luatexalignmark 1288
- \luatexalignntab 1289
- \luatexattribute 1290
- \luatexattributedef 1291

<code>\luatexbanner</code>	923		
<code>\luatexbodydir</code>	1327		
<code>\luatexboxdir</code>	1328		
<code>\luatexcatcodetable</code>	1292		
<code>\luatexclearmarks</code>	1293		
<code>\luatexcrampeddisplaystyle</code>	1294		
<code>\luatexcrampedscriptstyle</code> ..	1296		
<code>\luatexcrampedscriptstyle</code>	1297		
<code>\luatexcrampedtextstyle</code>	1298		
<code>\luatexfontid</code>	1299		
<code>\luatexformatname</code>	1300		
<code>\luatexgladers</code>	1301		
<code>\luatexinitcatcodetable</code>	1302		
<code>\luatexlualua</code>	1303		
<code>\luatexleftghost</code>	1329		
<code>\luatexlocalbrokenpenalty</code>	1330		
<code>\luatexlocalinterlinepenalty</code>	1332		
<code>\luatexlocalleftbox</code>	1333		
<code>\luatexlocalrightbox</code>	1334		
<code>\luatexluaescapestring</code>	1304		
<code>\luatexluafunction</code>	1305		
<code>\luatexmathdir</code>	1335		
<code>\luatexmathstyle</code>	1306		
<code>\luatexnokerns</code>	1307		
<code>\luatexnoligs</code>	1308		
<code>\luatexoutputbox</code>	1309		
<code>\luatexpagebottomoffset</code>	1336		
<code>\luatexpagedir</code>	1337		
<code>\luatexpageheight</code>	1338		
<code>\luatexpageleftoffset</code>	1310		
<code>\luatexpagerightoffset</code>	1339		
<code>\luatexpagetopoffset</code>	1311		
<code>\luatexpagewidth</code>	1340		
<code>\luatexpardir</code>	1341		
<code>\luatexpostexhyphenchar</code>	1312		
<code>\luatexposthyphenchar</code>	1313		
<code>\luatexpreeexhyphenchar</code>	1314		
<code>\luatexprehyphenchar</code>	1315		
<code>\luatexrevision</code>	924		
<code>\luatexrightghost</code>	1342		
<code>\luatexsavecatcodetable</code>	1316		
<code>\luatexscantexttokens</code>	1317		
<code>\luatexsuppressfontnotfounderror</code> ..	1287, 1326		
<code>\luatexsuppressifcsnameerror</code>	1319		
<code>\luatexsuppresslongerror</code>	1320		
<code>\luatexsuppressmathparerror</code>	1322		
<code>\luatexsuppressoutererror</code>	1323		
<code>\luatextextdir</code>	1343		
<code>\luatextracingfonts</code>	1283		
<code>\luatexUchar</code>	1324		
<code>\luatexversion</code>	45, 100, 925		
		M	
<code>\mag</code>	443		
<code>\mark</code>	444		
<code>\marks</code>	644		
math commands:			
<code>\c_math_subscript_token</code>			
.....	118, 474, 7416, 7474, 18309		
<code>\c_math_superscript_token</code>			
.....	118, 474, 7416, 7469, 18307		
<code>\c_math_toggle_token</code>			
.....	118, 473, 7416, 7450, 18303		
<code>\mathaccent</code>	445		
<code>\mathbin</code>	446		
<code>\mathchar</code>	447, 7600		
<code>\mathchardef</code>	448		
<code>\mathchoice</code>	449		
<code>\mathclose</code>	450		
<code>\mathcode</code>	451		
<code>\mathdelimitersmode</code>	926		
<code>\mathdir</code>	985		
<code>\mathdisplayskipmode</code>	927		
<code>\matheqnogapstep</code>	929		
<code>\mathinner</code>	452		
<code>\mathnolimitsmode</code>	930		
<code>\mathop</code>	453		
<code>\mathopen</code>	454		
<code>\mathoption</code>	931		
<code>\mathord</code>	455		
<code>\mathpenaltiesmode</code>	932		
<code>\mathpunct</code>	456		
<code>\mathrel</code>	457		
<code>\mathrulesfam</code>	933		
<code>\mathscriptboxmode</code>	935		
<code>\mathscriptsmode</code>	934		
<code>\mathstyle</code>	936		
<code>\mathsurround</code>	458		
<code>\mathsurroundmode</code>	937		
<code>\mathsurroundskip</code>	938		
max	191		
max commands:			
<code>\c_max_char_int</code>	87, 6136, 7299, 18847		
<code>\c_max_register_int</code>			
.....	87, 212, 788, 1454,		
.....	5261, 8796, 17658, 17684, 17721,		
.....	17729, 17733, 24202, 24232, 24234		
<code>\maxdeadcycles</code>	459		
<code>\maxdepth</code>	460		
<code>\mdfivesum</code>	873		
<code>\meaning</code>	461		
<code>\medmuskip</code>	462		
<code>\message</code>	463		
<code>\MessageBreak</code>	122		
meta commands:			
<code>.meta:n</code>	169, 11179		

- .meta:nn [169](#), [11181](#)
- \middle [645](#)
- min [191](#)
- minus commands:
 - \c_minus_inf_fp [186](#),
[195](#), [11598](#), [14642](#), [14726](#), [15578](#), [16356](#)
 - \c_minus_zero_fp
..... [185](#), [11598](#), [14638](#), [16995](#)
- \mkern [464](#)
- mm [195](#)
- mode commands:
 - \mode_if_horizontal:TF [99](#), [6484](#)
 - \mode_if_horizontal_p: [99](#), [6484](#)
 - \mode_if_inner:TF [99](#), [6486](#)
 - \mode_if_inner_p: [99](#), [6486](#)
 - \mode_if_math:TF [99](#), [6488](#)
 - \mode_if_math_p: [99](#), [6488](#)
 - \mode_if_vertical:TF [99](#), [6482](#)
 - \mode_if_vertical_p: [99](#), [6482](#)
 - \mode_leave_vertical:
..... [235](#), [22982](#), [23581](#)
- \month [465](#)
- \moveleft [466](#)
- \moveright [467](#)
- msg commands:
 - \msg_critical:nn [136](#), [8435](#)
 - \msg_critical:nn(nn) [238](#)
 - \msg_critical:nnn [136](#), [8435](#)
 - \msg_critical:nnnn [136](#), [8435](#)
 - \msg_critical:nnnnn [136](#), [8435](#)
 - \msg_critical:nnnnnn [136](#), [8435](#)
 - \msg_critical_text:n [135](#), [8354](#), [8438](#)
 - \msg_error:nn [137](#), [8446](#)
 - \msg_error:nnn [137](#), [8446](#)
 - \msg_error:nnnn [137](#), [8446](#)
 - \msg_error:nnnnn [137](#), [8446](#)
 - \msg_error:nnnnnn [137](#), [238](#), [8446](#)
 - \msg_error_text:n ... [135](#), [8354](#), [8453](#)
 - \msg_expandable_error:nn . [239](#), [24025](#)
 - \msg_expandable_error:nnn [239](#), [24025](#)
 - \msg_expandable_error:nnnn [239](#), [24025](#)
 - \msg_expandable_error:nnnnn
..... [239](#), [24025](#)
 - \msg_expandable_error:nnnnnn
..... [239](#), [24025](#)
 - \msg_fatal:nn [136](#), [8424](#)
 - \msg_fatal:nnn [136](#), [8424](#)
 - \msg_fatal:nnnn [136](#), [8424](#)
 - \msg_fatal:nnnnn [136](#), [8424](#)
 - \msg_fatal:nnnnnn [136](#), [8424](#)
 - \msg_fatal_text:n ... [135](#), [8354](#), [8427](#)
 - \msg_gset:nnn [134](#), [8211](#)
 - \msg_gset:nnnn [134](#), [8211](#)
 - \msg_if_exist:nnTF
..... [135](#), [8193](#), [8205](#), [8538](#)
 - \msg_if_exist_p:nn [135](#), [8193](#)
 - \msg_info:nn [137](#), [8475](#)
 - \msg_info:nnn [137](#), [8475](#)
 - \msg_info:nnnn [137](#), [8475](#)
 - \msg_info:nnnnn [137](#), [8475](#)
 - \msg_info:nnnnnn . [137](#), [137](#), [8475](#), [8711](#)
 - \msg_info_text:n [136](#), [8354](#), [8479](#)
 - \msg_interrupt:nnn
..... [139](#), [8275](#), [8426](#), [8437](#), [8452](#)
 - \msg_line_context: [135](#), [499](#), [2139](#),
[2148](#), [2937](#), [8201](#), [8268](#), [8357](#), [8362](#),
[8367](#), [8372](#), [8377](#), [8928](#), [8949](#), [10979](#)
 - \msg_line_number: ... [135](#), [8268](#), [10769](#)
 - \msg_log:n [140](#), [8339](#), [8477](#)
 - \msg_log:nn [137](#), [8483](#)
 - \msg_log:nnn [137](#), [8483](#)
 - \msg_log:nnnn [137](#), [8483](#)
 - \msg_log:nnnnn [137](#), [8483](#)
 - \msg_log:nnnnnn
..... [137](#), [5242](#), [7127](#), [7140](#), [8174](#),
[8483](#), [9166](#), [9321](#), [9884](#), [11510](#), [23209](#)
 - \msg_log_eval:Nn . [239](#), [6112](#), [6283](#),
[10379](#), [10514](#), [10616](#), [13880](#), [24052](#)
 - \msg_new:nnn [134](#), [8211](#), [8662](#)
 - \msg_new:nnnn ... [134](#), [498](#), [8211](#), [8660](#)
 - \msg_none:nn [138](#), [8489](#)
 - \msg_none:nnn [138](#), [8489](#)
 - \msg_none:nnnn [138](#), [8489](#)
 - \msg_none:nnnnn [138](#), [8489](#)
 - \msg_none:nnnnnn [138](#), [8489](#)
 - \msg_redirect_class:nn [138](#), [8611](#)
 - \msg_redirect_module:nnn .. [139](#), [8611](#)
 - \msg_redirect_name:nnn [139](#), [8602](#)
 - \msg_see_documentation_text:n ...
..... [136](#), [8379](#), [8430](#), [8441](#), [8456](#)
 - \msg_set:nnn [134](#), [8211](#), [8666](#)
 - \msg_set:nnnn [134](#), [8211](#), [8664](#)
 - \msg_show:nn [239](#), [8490](#)
 - \msg_show:nnn [239](#), [8490](#)
 - \msg_show:nnnn [239](#), [8490](#)
 - \msg_show:nnnnn [239](#), [8490](#)
 - \msg_show:nnnnnn [239](#), [239](#),
[408](#), [465](#), [496](#), [5240](#), [7125](#), [7139](#),
[8172](#), [8490](#), [9165](#), [9320](#), [9883](#), [11508](#),
[18404](#), [18412](#), [21133](#), [21142](#), [23206](#)
 - \msg_show_eval:Nn [239](#), [6108](#), [6281](#),
[10375](#), [10510](#), [10612](#), [13878](#), [24052](#)
 - \msg_show_item:n
... [239](#), [239](#), [5250](#), [7135](#), [7144](#), [24057](#)
 - \msg_show_item:nn [239](#), [496](#), [8182](#), [24057](#)
 - \msg_show_item_unbraced:n [239](#), [24057](#)

- \msg_show_item_unbraced:nn . [239](#),
[522](#), [9173](#), [9328](#), [11518](#), [23225](#), [24057](#)
- \msg_term:n [140](#), [8339](#), [8469](#)
- \msg_warning:nn [137](#), [8467](#)
- \msg_warning:nnn [137](#), [8467](#)
- \msg_warning:nnnn [137](#), [8467](#)
- \msg_warning:nnnnn [137](#), [8467](#)
- \msg_warning:nnnnnn [137](#), [8467](#), [8710](#)
- \msg_warning_text:n [135](#), [8354](#), [8471](#)
- msg internal commands:
- __msg_chk_free:nn [8203](#), [8213](#)
- __msg_chk_if_free:nn [8198](#)
- __msg_class_chk_exist:nTF
. [8525](#), [8540](#), [8607](#), [8617](#), [8622](#)
- \l__msg_class_loop_seq [509](#), [8534](#),
[8626](#), [8634](#), [8644](#), [8645](#), [8648](#), [8650](#)
- __msg_class_new:nn
. [506](#), [510](#), [8385](#), [8424](#), [8435](#),
[8446](#), [8467](#), [8475](#), [8483](#), [8489](#), [8490](#)
- \l__msg_class_tl [507](#),
[509](#), [8530](#), [8547](#), [8560](#), [8581](#), [8585](#),
[8588](#), [8596](#), [8635](#), [8637](#), [8639](#), [8653](#)
- \c__msg_coding_error_text_tl
. [8236](#), [8716](#), [8724](#), [8750](#),
[8768](#), [8777](#), [8789](#), [8803](#), [8812](#), [8833](#),
[8842](#), [8849](#), [8858](#), [8864](#), [8871](#), [8878](#),
[8885](#), [8892](#), [8900](#), [8908](#), [8938](#), [8951](#)
- \c__msg_continue_text_tl [8236](#), [8280](#)
- \c__msg_critical_text_tl [8236](#), [8443](#)
- \l__msg_current_class_tl
. [509](#), [8530](#), [8542](#),
[8580](#), [8585](#), [8588](#), [8596](#), [8625](#), [8639](#)
- __msg_error:Nnnnnn [8446](#)
- __msg_error_code:nnnnnn [8709](#)
- __msg_expandable_error:n
. [518](#), [9031](#), [9050](#)
- __msg_expandable_error:w [518](#), [9031](#)
- __msg_expandable_error_module:nn
. [24025](#)
- __msg_fatal_code:nnnnnn [8705](#)
- \c__msg_fatal_text_tl [8236](#), [8432](#)
- \c__msg_help_text_tl [8236](#), [8284](#)
- \l__msg_hierarchy_seq
. [507](#), [508](#), [8533](#), [8563](#), [8573](#), [8578](#)
- \l__msg_internal_tl [8189](#), [8514](#), [8520](#)
- __msg_interrupt_more_text:n
. [500](#), [8287](#)
- __msg_interrupt_text:n [8290](#), [8301](#)
- __msg_interrupt_wrap:nn
. [8279](#), [8283](#), [8287](#)
- __msg_kernel_class_new:nN
. [511](#), [8667](#), [8705](#), [8709](#), [8710](#), [8711](#)
- __msg_kernel_class_new_aux:nN [8667](#)
- \l__msg_line_context_bool
. [8190](#), [8357](#), [8362](#), [8367](#), [8372](#), [8377](#)
- \c__msg_more_text_prefix_tl
. [8191](#), [8222](#), [8231](#), [8449](#)
- \c__msg_no_info_text_tl [8236](#), [8279](#)
- __msg_no_more_text:nnnn [8446](#)
- \c__msg_on_line_text_tl [8236](#), [8271](#)
- __msg_redirect:nnn [8611](#)
- __msg_redirect_loop_chk:nnn
. [8611](#), [8653](#)
- __msg_redirect_loop_list:n [8611](#)
- \l__msg_redirect_prop
. [8532](#), [8560](#), [8605](#), [8608](#)
- \c__msg_return_text_tl
. [8236](#), [8719](#), [8727](#), [8734](#)
- __msg_show:n [505](#), [8490](#)
- __msg_show:nn [8490](#)
- __msg_show:w [8490](#)
- __msg_show_dot:w [8490](#)
- __msg_show_eval:nnN [24052](#)
- \c__msg_text_prefix_tl [518](#), [8191](#),
[8195](#), [8220](#), [8229](#), [8429](#), [8440](#), [8455](#),
[8472](#), [8480](#), [8486](#), [8493](#), [9053](#), [24030](#)
- __msg_tmp:w [9032](#), [9045](#)
- \c__msg_trouble_text_tl [8236](#)
- __msg_use:nnnnnnn [8395](#), [8535](#)
- __msg_use_code: [507](#), [8535](#)
- __msg_use_hierarchy:nwN [8535](#)
- __msg_use_redirect_module:n
. [508](#), [8535](#)
- __msg_use_redirect_name:n [8535](#)
- \mskip [468](#)
- \muexpr [646](#)
- multichoice commands:
- .multichoice: [169](#), [11183](#)
- multichoices commands:
- .multichoices:nn [169](#), [11183](#)
- \multiply [469](#)
- \mskip [470](#), [7608](#)
- muskip commands:
- \c_max_muskip [164](#), [10617](#)
- \muskip_add:Nn [162](#), [10582](#)
- \muskip_const:Nn
. [162](#), [10529](#), [10617](#), [10618](#)
- \muskip_eval:n
[163](#), [163](#), [10533](#), [10598](#), [10612](#), [10616](#)
- \muskip_gadd:Nn [162](#), [10582](#)
- \muskip_gset:Nn [162](#), [10568](#)
- \muskip_gset_eq:NN [162](#), [10576](#)
- \muskip_gsub:Nn [163](#), [10582](#)
- \muskip_gzero:N [162](#), [10536](#), [10547](#)
- \muskip_gzero_new:N [162](#), [10544](#)
- \muskip_if_exist:NTF
. [162](#), [10545](#), [10547](#), [10550](#)

<code>\muskip_if_exist_p:N</code>	162, 10550	<code>\normallanguage</code>	1377
<code>\muskip_log:N</code>	163, 10613	<code>\normalleft</code>	1394, 1395
<code>\muskip_log:n</code>	163, 10613	<code>\normalmathop</code>	1378
<code>\muskip_new:N</code>	162, 162, 10521, 10532, 10545, 10547, 10619, 10620, 10621, 10622	<code>\normalmiddle</code>	1396
<code>\muskip_set:Nn</code>	162, 10568	<code>\normalmonth</code>	1379
<code>\muskip_set_eq:NN</code>	162, 10576	<code>\normalouter</code>	1380
<code>\muskip_show:N</code>	163, 10609	<code>\normalover</code>	1381
<code>\muskip_show:n</code>	163, 564, 10611	<code>\normalright</code>	1397
<code>\muskip_sub:Nn</code>	163, 10582	<code>\normalshowtokens</code>	1390
<code>\muskip_use:N</code>	163, 163, 10606, 10607	<code>\normalunexpanded</code>	1383
<code>\muskip_zero:N</code>	162, 162, 10536, 10545	<code>\normalvcenter</code>	1382
<code>\muskip_zero_new:N</code>	162, 10544	<code>\normalvoffset</code>	1389
<code>\g_tmpa_muskip</code>	164, 10619	<code>\nospaces</code>	942
<code>\l_tmpa_muskip</code>	164, 10619	notexpanded commands:	
<code>\g_tmpb_muskip</code>	164, 10619	<code>\notexpanded: <token></code>	126
<code>\l_tmpb_muskip</code>	164, 10619	<code>\novrule</code>	943
<code>\c_zero_muskip</code>	164, 10538, 10541, 10617	<code>\nulldelimiterspace</code>	480
<code>\muskipdef</code>	471	<code>\nullfont</code>	481
<code>\mutoglu</code>	647	<code>\num</code>	179
		<code>\number</code>	55, 482
		<code>\numexpr</code>	167, 181, 648

N

<code>nan</code>	195
<code>nc</code>	195
<code>nd</code>	195
<code>\newbox</code>	412
<code>\newcatcodetable</code>	52
<code>\newcount</code>	412
<code>\newdimen</code>	412
<code>\newlinechar</code>	103, 472
<code>\next</code>	74, 106, 125, 125, 125, 131, 140, 144, 147, 155
<code>\NG</code>	25306
<code>\ng</code>	25306
<code>\noalign</code>	473
<code>\noautospacing</code>	1213
<code>\noautoxspacing</code>	1214
<code>\noboundary</code>	474
<code>\noexpand</code>	118, 122, 133, 136, 475
<code>\nohrule</code>	939
<code>\noindent</code>	476
<code>\nokerns</code>	940
<code>\noligs</code>	941
<code>\nolimits</code>	477
<code>\nonscript</code>	478
<code>\nonstopmode</code>	479
<code>\normaldeviate</code>	1007
<code>\normalend</code>	1373, 1374, 9096, 9263
<code>\normaleveryjob</code>	1375
<code>\normalexpanded</code>	1384
<code>\normalhoffset</code>	1387
<code>\normalinput</code>	1376
<code>\normalitaliccorrection</code>	1386, 1388

O

<code>\O</code>	25307
<code>\o</code>	25307
<code>\OE</code>	25308
<code>\oe</code>	25308
<code>\omit</code>	483
one commands:	
<code>\c_minus_one</code>	433, 6150
<code>\c_one_degree_fp</code>	186, 195, 13197, 13883
<code>\openin</code>	484
<code>\openout</code>	485
<code>\or</code>	486
or commands:	
<code>\or:</code>	89, 382, 384, 603, 737, 1402, 2238, 2239, 2240, 2241, 2242, 2243, 2244, 2245, 2246, 4366, 4442, 5261, 5895, 5896, 5897, 5898, 5899, 5900, 5901, 5902, 5903, 5904, 5905, 5906, 5907, 5908, 5909, 5910, 5911, 5912, 5913, 5914, 5915, 5916, 5917, 5918, 5919, 5928, 5929, 5930, 5931, 5932, 5933, 5934, 5935, 5936, 5937, 5938, 5939, 5940, 5941, 5942, 5943, 5944, 5945, 5946, 5947, 5948, 5949, 5950, 5951, 5952, 7348, 7352, 7355, 7359, 7363, 7365, 7367, 7369, 7370, 7372, 7374, 7376, 7378, 9527, 9528, 9529, 9530, 9531, 9532, 9533, 11637, 11638, 11639, 11888, 11903, 11904, 12300, 12301, 12326, 13580, 13581, 13582, 13618, 14338, 14339, 14340, 14463,

14548, 14634, 14635, 14636, 14637,	\pagetopoffset	948
14638, 14639, 14640, 14641, 14642,	\pagetotal	501
14721, 14724, 15059, 15060, 15074,	\pagewidth	1010
15358, 15579, 15604, 15610, 15611,	\par 9, 10, 10, 10, 11, 11, 11, 12, 12, 12,	
15612, 15613, 15614, 15763, 15798,	13, 13, 13, 143, 318, 502, 916, 23257	
15800, 15808, 15895, 15899, 15900,	\pardir	987
15901, 15902, 15903, 15904, 15905,	\parfillskip	503
15906, 15907, 15908, 15915, 15916,	\parindent	504
15917, 15918, 15919, 15920, 15921,	\parshape	505
15922, 15923, 15924, 15931, 15932,	\parshapedimen	650
15933, 15934, 15935, 15936, 15937,	\parshapeindent	651
15938, 15939, 15940, 15947, 15948,	\parshapelength	652
15949, 15950, 15951, 15952, 15953,	\parskip	506
15954, 15955, 15956, 15963, 15964,	\patterns	507
15965, 15966, 15967, 15968, 15969,	\pausing	508
15970, 15971, 15972, 15979, 15980,	pc	195
15981, 15982, 15983, 15984, 15985,	\pdfadjustspacing	750
15986, 15987, 15988, 16006, 16057,	\pdfannot	675
16060, 16069, 16180, 16203, 16204,	\pdfcatalog	676
16265, 16270, 16280, 16285, 16295,	\pdfcolorstack	678
16300, 16310, 16315, 16325, 16330,	\pdfcolorstackinit	679
16340, 16345, 16792, 16793, 16838,	\pdfcompresslevel	677
16923, 16926, 16938, 16944, 16991,	\pdfcopyfont	751
16993, 16994, 17004, 17010, 17087,	\pdfcreationdate	680
17088, 17095, 17141, 17142, 17149,	\pdfdecimaldigits	681
17215, 17216, 17457, 18155, 18156,	\pdfdest	682
18349, 18350, 18646, 18647, 18648,	\pdfdestmargin	683
18649, 18921, 18922, 18923, 18924,	\pdfdraftmode	752
18925, 20228, 20285, 20634, 20635	\pdfeachlinedepth	753
\outer	5, 412, 487	
\output	488	
\outputbox	944	
\outputmode	1008	
\outputpenalty	489	
\over	490	
\overfullrule	491	
\overline	492	
\overwithdelims	493	
P		
\PackageError	125, 133	
\pagebottomoffset	945	
\pagedepth	494	
\pagedir	986	
\pagediscards	649	
\pagefilllstretch	495	
\pagefillstretch	496	
\pagefilstretch	497	
\pagegoal	498	
\pageheight	1009	
\pageleftoffset	946	
\pagerightoffset	947	
\pageshrink	499	
\pagestretch	500	
\pdfadjustspacing	750	
\pdfannot	675	
\pdfcatalog	676	
\pdfcolorstack	678	
\pdfcolorstackinit	679	
\pdfcompresslevel	677	
\pdfcopyfont	751	
\pdfcreationdate	680	
\pdfdecimaldigits	681	
\pdfdest	682	
\pdfdestmargin	683	
\pdfdraftmode	752	
\pdfeachlinedepth	753	
\pdfeachlineheight	754	
\pdfendlink	684	
\pdfendthread	685	
\pdfextension	949	
\pdffeedback	950	
\pdfffilemoddate	755	
\pdfffilesize	756	
\pdfffirstlineheight	757	
\pdfffontattr	686	
\pdfffontexpand	758	
\pdfffontname	687	
\pdfffontobjnum	688	
\pdfffontsize	759	
\pdfgamma	689	
\pdfgentounicode	692	
\pdfglyphtounicode	693	
\pdfhorigin	694	
\pdfignoreddimen	760	
\pdfimageapplygamma	690	
\pdfimagegamma	691	
\pdfimagehicolor	695	
\pdfimageresolution	696	
\pdfincludechars	697	
\pdfinclusioncopyfonts	698	
\pdfinclusionerrorlevel	700	

<code>\pdfinfo</code>	702	<code>\pdfTEX_copyfont:D</code>	751, 991
<code>\pdfinsertht</code>	761	<code>\pdfTEX_draftmode:D</code>	752, 992
<code>\pdflastannot</code>	703	<code>\pdfTEX_eachlinedepth:D</code>	753
<code>\pdflastlinedepth</code>	762	<code>\pdfTEX_eachlineheight:D</code>	754
<code>\pdflastlink</code>	704	<code>\pdfTEX_efcode:D</code>	787
<code>\pdflastobj</code>	705	<code>\pdfTEX_filemoddate:D</code>	755
<code>\pdflastxform</code>	706	<code>\pdfTEX_filesize:D</code>	756
<code>\pdflastximage</code>	707	<code>\pdfTEX_firstlineheight:D</code>	757
<code>\pdflastximagecolordepth</code>	708	<code>\pdfTEX_fontexpand:D</code>	758, 993
<code>\pdflastximagepages</code>	710	<code>\pdfTEX_fontsize:D</code>	759
<code>\pdflastxpos</code>	763	<code>\pdfTEX_if_engine:TF</code>	
<code>\pdflastypos</code>	764	27196, 27198, 27200
<code>\pdflinkmargin</code>	711	<code>\pdfTEX_if_engine_p:</code>	27194
<code>\pdfliteral</code>	712	<code>\pdfTEX_ifabsdim:D</code>	747, 994
<code>\pdfmapfile</code>	765	<code>\pdfTEX_ifabsnum:D</code>	748, 995
<code>\pdfmapline</code>	766	<code>\pdfTEX_ifincsname:D</code>	788
<code>\pdfmdfivesum</code>	767	<code>\pdfTEX_ifprimitive:D</code>	749, 874
<code>\pdfminorversion</code>	713	<code>\pdfTEX_ignoredimen:D</code>	760
<code>\pdfnames</code>	714	<code>\pdfTEX_ignoreligaturesinfont:D</code>	997
<code>\pdfnoligatures</code>	768	<code>\pdfTEX_insertht:D</code>	761, 998
<code>\pdfnormaldeviate</code>	769	<code>\pdfTEX_lastlinedepth:D</code>	762
<code>\pdfobj</code>	715	<code>\pdfTEX_lastxpos:D</code>	763, 1005
<code>\pdfobjcompresslevel</code>	716	<code>\pdfTEX_lastypos:D</code>	764, 1006
<code>\pdfoutline</code>	718	<code>\pdfTEX_leftmarginkern:D</code>	789
<code>\pdfoutput</code>	719	<code>\pdfTEX_letterspacefont:D</code>	790
<code>\pdfpageattr</code>	720	<code>\pdfTEX_lpcode:D</code>	791
<code>\pdfpagebox</code>	721	<code>\pdfTEX_mapfile:D</code>	765, 1349
<code>\pdfpageheight</code>	770	<code>\pdfTEX_mapline:D</code>	766, 1350
<code>\pdfpageref</code>	722	<code>\pdfTEX_mdfivesum:D</code> ..	767, 873, 23980
<code>\pdfpageresources</code>	723	<code>\pdfTEX_noligatures:D</code>	768
<code>\pdfpagesattr</code>	724	<code>\pdfTEX_normaldeviate:D</code> ..	769, 1007
<code>\pdfpagewidth</code>	771	<code>\pdfTEX_pageheight:D</code> ..	770, 1009, 1338
<code>\pdfpkmode</code>	772	<code>\pdfTEX_pagewidth:D</code>	771, 1340
<code>\pdfpkresolution</code>	773	<code>\pdfTEX_pagewith:D</code>	1010
<code>\pdfprimitive</code>	774	<code>\pdfTEX_pdfannot:D</code>	675
<code>\pdfprotrudechars</code>	775	<code>\pdfTEX_pdfcatalog:D</code>	676
<code>\pdfpxdimen</code>	776	<code>\pdfTEX_pdfcolorstack:D</code>	
<code>\pdfrandomseed</code>	777	678, 25837, 25846
<code>\pdfrefobj</code>	725	<code>\pdfTEX_pdfcolorstackinit:D</code> ..	679
<code>\pdfrefxform</code>	726	<code>\pdfTEX_pdfcompresslevel:D</code>	677
<code>\pdfrefximage</code>	727	<code>\pdfTEX_pdfcreationdate:D</code>	680
<code>\pdfrestore</code>	728	<code>\pdfTEX_pdfdecimaldigits:D</code>	681
<code>\pdfretval</code>	729	<code>\pdfTEX_pdfdest:D</code>	682
<code>\pdfsave</code>	730	<code>\pdfTEX_pdfdestmargin:D</code>	683
<code>\pdfsavepos</code>	778	<code>\pdfTEX_pdfendlink:D</code>	684
<code>\pdfsetmatrix</code>	731	<code>\pdfTEX_pdfendthread:D</code>	685
<code>\pdfsetrandomseed</code>	780	<code>\pdfTEX_pdffontattr:D</code>	686
<code>\pdfshellescape</code>	781	<code>\pdfTEX_pdffontname:D</code>	687
<code>\pdfstartlink</code>	732	<code>\pdfTEX_pdffontobjnum:D</code>	688
<code>\pdfstartthread</code>	733	<code>\pdfTEX_pdfgamma:D</code>	689
<code>\pdfstrcmp</code>	40, 376, 779	<code>\pdfTEX_pdfgentounicode:D</code>	692
<code>\pdfsuppressptexinfo</code>	734	<code>\pdfTEX_pdfglyptounicode:D</code> ..	693
pdfTeX commands:		<code>\pdfTEX_pdfhorigin:D</code>	694
<code>\pdfTEX_adjustspacing:D</code>	750, 990	<code>\pdfTEX_pdfimageapplygamma:D</code> ..	690

- \pdfutex_pdfimagegamma:D 691
- \pdfutex_pdfimagehicolor:D 695
- \pdfutex_pdfimageresolution:D .. 696
- \pdfutex_pdfincludechars:D 697
- \pdfutex_pdfinclusioncopyfonts:D 699
- \pdfutex_pdfinclusionerrorlevel:D 701
- \pdfutex_pdfinfo:D 702
- \pdfutex_pdflastannot:D 703
- \pdfutex_pdflastobj:D 705
- \pdfutex_pdflastxform:D 706, 1000
- \pdfutex_pdflastximage:D
..... 707, 1002, 26312, 26316
- \pdfutex_pdflastximagecolordepth:D
..... 709
- \pdfutex_pdflastximagepages:D ...
..... 710, 1004
- \pdfutex_pdflinkmargin:D 711
- \pdfutex_pdfliteral:D 712, 26173
- \pdfutex_pdfminorversion:D 713
- \pdfutex_pdfnames:D 714
- \pdfutex_pdfobj:D 715
- \pdfutex_pdfobjcompresslevel:D .. 717
- \pdfutex_pdfoutline:D 718
- \pdfutex_pdfoutput:D .. 719, 1008, 6565
- \pdfutex_pdfpageattr:D 720
- \pdfutex_pdfpagebox:D 721
- \pdfutex_pdfpageref:D 722
- \pdfutex_pdfpageresources:D 723
- \pdfutex_pdfpagesattr:D 724
- \pdfutex_pdfrefobj:D 725
- \pdfutex_pdfrefxform:D 726, 1014
- \pdfutex_pdfrefximage:D
..... 727, 1015, 26312, 26321
- \pdfutex_pdfrestore:D 728, 26187
- \pdfutex_pdfretval:D 729
- \pdfutex_pdfsave:D 730, 26181
- \pdfutex_pdfsetmatrix:D ... 731, 26193
- \pdfutex_pdfstartlink:D 732
- \pdfutex_pdfstartthread:D 733
- \pdfutex_pdfsuppressptexinfo:D .. 735
- \pdfutex_pdftexbanner:D 784, 1367
- \pdfutex_pdftexrevision:D .. 785, 1368
- \pdfutex_pdftexversion:D
..... 287, 786, 1346, 1369, 6546
- \pdfutex_pdfthread:D 736
- \pdfutex_pdfthreadmargin:D 737
- \pdfutex_pdftrailer:D 738
- \pdfutex_pdfuniqueresname:D 739
- \pdfutex_pdfvorigin:D 740
- \pdfutex_pdfxform:D 741, 1017
- \pdfutex_pdfxformattr:D 742
- \pdfutex_pdfxformname:D 743
- \pdfutex_pdfxformresources:D ... 744
- \pdfutex_pdfximage:D . 745, 1018, 26293
- \pdfutex_pdfximagebbox:D .. 746, 26287
- \pdfutex_pkmode:D 772
- \pdfutex_pkresolution:D 773
- \pdfutex_primitive:D 774, 875
- \pdfutex_protrudechars:D ... 775, 1011
- \pdfutex_pxdimen:D 776, 1012
- \pdfutex_quitvmode:D 792, 23583, 23584
- \pdfutex_randomseed:D
..... 777, 1013, 24260, 24261
- \pdfutex_rightmarginkern:D 793
- \pdfutex_rpcode:D 794
- \pdfutex_savepos:D 778, 1016
- \pdfutex_setrandomseed:D
..... 780, 1019, 24264
- \pdfutex_shellescape:D 781, 876, 24273
- \pdfutex_strcmp:D 779, 4076, 11949
- \pdfutex_synctex:D 795
- \pdfutex_tagcode:D 796
- \pdfutex_tracingfonts:D
..... 782, 1020, 1277, 1279, 1283
- \pdfutex_uniformdeviate:D ... 779,
779, 783, 976, 1021, 17350, 17366,
17369, 17548, 24193, 24224, 24259
- \pdfutxbanner 784
- \pdfutxrevision 785
- \pdfutxversion 95, 786
- \pdfutxthread 736
- \pdfutxthreadmargin 737
- \pdfutracingfonts 782, 1278, 1279
- \pdfutrailer 738
- \pdfuniformdeviate 783
- \pdfuniqueresname 739
- \pdfvariable 951
- \pdfvorigin 740
- \pdfxform 741
- \pdfxformattr 742
- \pdfxformname 743
- \pdfxformresources 744
- \pdfximage 745
- \pdfximagebbox 746
- peek commands:
 - \peek_after:Nw
.. 101, 122, 122, 122, 7682, 7707, 7765
 - \peek_catcode:Ntf 122, 7789
 - \peek_catcode_ignore_spaces:Ntf ..
..... 123, 7789
 - \peek_catcode_remove:Ntf .. 123, 7789
 - \peek_catcode_remove_ignore_
spaces:Ntf 123, 7789
 - \peek_charcode:Ntf 123, 7805
 - \peek_charcode_ignore_spaces:Ntf
..... 123, 7805
 - \peek_charcode_remove:Ntf . 123, 7805

- \peek_charcode_remove_ignore_spaces:NTF [124](#), [7805](#)
- \peek_gafter:Nw [122](#), [122](#), [7682](#)
- \peek_meaning:NTF [124](#), [7821](#)
- \peek_meaning_ignore_spaces:NTF .
..... [124](#), [7821](#)
- \peek_meaning_remove:NTF .. [124](#), [7821](#)
- \peek_meaning_remove_ignore_spaces:NTF [124](#), [7821](#)
- \peek_N_type:TF
..... [251](#), [25630](#), [25667](#), [25669](#)
- peek internal commands:
 - __peek_def:nnnn [7772](#),
[7789](#), [7793](#), [7797](#), [7801](#), [7805](#), [7809](#),
[7813](#), [7817](#), [7821](#), [7825](#), [7829](#), [7833](#)
 - __peek_def:nnnnn [7772](#)
 - __peek_execute_branches:
..... [483](#), [7769](#), [7784](#)
 - __peek_execute_branches_catcode: [7729](#), [7792](#), [7794](#), [7800](#), [7802](#)
 - __peek_execute_branches_catcode_aux: [7729](#)
 - __peek_execute_branches_catcode_auxii:N [7729](#)
 - __peek_execute_branches_catcode_auxiii: [7729](#)
 - __peek_execute_branches_charcode:
..... [7729](#), [7808](#), [7810](#), [7816](#), [7818](#)
 - __peek_execute_branches_meaning: [7721](#), [7824](#), [7826](#), [7832](#), [7834](#)
 - __peek_execute_branches_N_type: [25630](#)
 - __peek_false:w [1011](#), [7678](#),
[7702](#), [7726](#), [7749](#), [7759](#), [25647](#), [25660](#)
 - __peek_get_prefix_arg_replacement:wN [7838](#)
 - __peek_ignore_spaces_execute_branches: [7762](#),
[7796](#), [7804](#), [7812](#), [7820](#), [7828](#), [7836](#)
 - __peek_N_type:w [25630](#)
 - __peek_N_type_aux:nnw [25630](#)
 - \l_peek_search_tl
.... [481](#), [483](#), [7677](#), [7695](#), [7746](#), [7756](#)
 - \l_peek_search_token
..... [481](#), [7676](#), [7694](#), [7723](#)
 - __peek_tmp:w [7678](#), [7689](#), [25631](#), [25653](#)
 - __peek_token_generic:NNTF [1011](#),
[7709](#), [7711](#), [7713](#), [25664](#), [25668](#), [25670](#)
 - __peek_token_generic_aux:NNTF .
..... [7691](#), [7710](#), [7716](#)
 - __peek_token_remove_generic:NNTF [7709](#), [7717](#), [7719](#)
 - __peek_true:w . [1011](#), [7678](#), [7701](#),
[7724](#), [7747](#), [7757](#), [25645](#), [25659](#), [25660](#)
 - __peek_true_aux:w
..... [481](#), [7678](#), [7688](#), [7696](#), [7710](#)
 - __peek_true_remove:w [481](#), [7686](#), [7716](#)
 - \penalty 509
 - pdfTeX commands:
 - \pdfTeX_pdflastlink:D 704
 - \pi 12604, 12605
 - pi 195
 - \pm 14090, 14091
 - \postbreakpenalty 1215
 - \postdisplaypenalty 510
 - \postexhyphenchar 952
 - \posthyphenchar 953
 - \prebinoppenalty 954
 - \prebreakpenalty 1216
 - \predisplaydirection 653
 - \predisplaygapfactor 955
 - \predisplaypenalty 511
 - \predisplaysize 512
 - \preexhyphenchar 957
 - \prehyphenchar 958
 - \prerelpenalty 959
 - \pretolerance 513
 - \prevdepth 514
 - \prevgraf 515
 - prg commands:
 - \prg_break: [100](#), [404](#), [405](#),
[495](#), [882](#), [973](#), [2457](#), [3925](#), [5069](#),
[5096](#), [5695](#), [6496](#), [6503](#), [8122](#), [8143](#),
[9776](#), [11699](#), [11708](#), [13704](#), [13724](#),
[13725](#), [13986](#), [13987](#), [14000](#), [14100](#),
[14101](#), [14102](#), [17325](#), [17929](#), [18004](#),
[18344](#), [18418](#), [18448](#), [18449](#), [18450](#),
[18451](#), [18452](#), [18453](#), [18821](#), [18825](#),
[20742](#), [20769](#), [24097](#), [24130](#), [24136](#)
 - \prg_break:n
. [100](#), [100](#), [2457](#), [3927](#), [4943](#), [5082](#),
[5705](#), [6496](#), [6505](#), [8027](#), [11715](#), [20308](#)
 - \prg_break_point:
..... [100](#), [100](#), [539](#), [790](#),
[790](#), [797](#), [975](#), [2457](#), [3915](#), [4940](#),
[5070](#), [5096](#), [5162](#), [5169](#), [5700](#), [6496](#),
[6498](#), [6499](#), [8022](#), [8107](#), [8143](#), [9753](#),
[11700](#), [11709](#), [13705](#), [13726](#), [13988](#),
[14104](#), [17326](#), [17769](#), [17810](#), [17922](#),
[17929](#), [18267](#), [18419](#), [18455](#), [18799](#),
[20309](#), [20609](#), [20763](#), [24097](#), [24131](#)
 - \prg_break_point:Nn . [62](#), [100](#), [100](#),
[325](#), [404](#), [422](#), [446](#), [554](#), [2448](#), [3556](#),
[3574](#), [3584](#), [4201](#), [4227](#), [4247](#), [5097](#),
[5132](#), [5143](#), [5747](#), [6496](#), [6496](#), [6497](#),

- 6502, 6936, 6950, 6970, 6988, 8144,
8160, 9223, 10327, 14173, 18396, 24098
- \prg_do_nothing: 8,
100, 349, 395, 449, 464, 610, 776,
838, 885, 2446, 2457, 2842, 2867,
3234, 3263, 3313, 3328, 4017, 4019,
4791, 4798, 5035, 5037, 6582, 6588,
6596, 6750, 6949, 6957, 7106, 7110,
7117, 9252, 9526, 10701, 11939,
12006, 12040, 12066, 12074, 13589,
17307, 17856, 18002, 18003, 18238,
18287, 19108, 19151, 19152, 19159,
19160, 20837, 21002, 24378, 24702
- \prg_generate_conditional_-
variant:Nnn
..... 237, 2982, 3416, 3426,
3437, 3458, 3478, 3488, 3548, 3813,
4105, 4118, 4126, 4157, 4670, 4684,
4923, 4944, 5038, 5040, 5054, 5056,
5058, 5060, 6279, 6768, 6782, 6783,
6926, 6928, 8051, 8052, 8100, 8124,
8135, 9125, 21836, 21838, 21842, 22440
- \prg_map_break:Nn
..... 100, 100, 325, 361, 446,
461, 496, 2448, 3597, 3599, 4260,
4262, 5087, 5089, 6496, 6500, 6502,
7005, 7007, 8169, 8171, 9206, 9208
- \prg_new_conditional:Nnn
..... 92, 1863, 6240
- \prg_new_conditional:Npnn
..... 92, 92, 237, 291, 292, 305,
474, 484, 1846, 2387, 3414, 3418,
3428, 3448, 3450, 3492, 3508, 3798,
3815, 3834, 3869, 3886, 3897, 4098,
4107, 4112, 4654, 4662, 4672, 4678,
4915, 5491, 5550, 5590, 5600, 6205,
6211, 6240, 6271, 6303, 6358, 6367,
6373, 6382, 6482, 6484, 6486, 6488,
6598, 6879, 7438, 7443, 7448, 7453,
7460, 7466, 7472, 7477, 7482, 7487,
7492, 7497, 7502, 7507, 7514, 7529,
7534, 7569, 7615, 8095, 8102, 8193,
9178, 10128, 10133, 10454, 10469,
11492, 11499, 11883, 13045, 13898,
13914, 18913, 18933, 18955, 19013,
21832, 21834, 21840, 22430, 24311
- \prg_new_eq_conditional:NNn
..... 93, 1959, 3127,
3128, 4068, 4070, 4072, 4074, 4820,
4822, 5234, 5235, 5236, 5237, 5238,
5239, 5421, 5423, 6240, 6299, 6301,
6686, 6688, 6875, 6877, 8091, 8093,
10025, 10027, 10414, 10416, 10550,
10552, 13896, 13897, 21798, 21800
- \prg_new_protected_conditional:Nnn
..... 92, 1863, 6240
- \prg_new_protected_conditional:Npnn
..... 92, 1846, 3460, 3480, 4120, 4128,
4925, 5034, 5036, 5042, 5045, 5048,
5051, 6240, 6759, 6769, 6771, 6893,
6897, 8031, 8041, 8126, 9115, 9778,
19337, 21147, 21152, 21165, 21167
- \prg_replicate:nn
..... 99, 443, 6434, 8992, 9378,
15300, 16156, 16390, 16594, 16595,
16597, 16599, 16612, 16649, 17172,
17180, 18523, 19114, 19850, 20235,
20261, 20414, 20422, 20842, 21310,
21315, 21322, 21425, 21430, 26896
- \prg_return_false: 93,
94, 368, 400, 416, 458, 459, 897,
1842, 2053, 2058, 2071, 2076, 2084,
2101, 2390, 3423, 3434, 3445, 3455,
3470, 3485, 3499, 3515, 3810, 3831,
3847, 3855, 3865, 3878, 3892, 3906,
4103, 4110, 4116, 4124, 4132, 4659,
4667, 4692, 4920, 4939, 4958, 5489,
5521, 5526, 5555, 5595, 5603, 6208,
6216, 6240, 6276, 6308, 6364, 6370,
6371, 6377, 6386, 6483, 6485, 6487,
6489, 6613, 6616, 6762, 6776, 6882,
6917, 6923, 7441, 7446, 7451, 7456,
7463, 7470, 7475, 7480, 7485, 7490,
7495, 7500, 7505, 7510, 7527, 7532,
7537, 7542, 7575, 7578, 7590, 7619,
7644, 7661, 7670, 8039, 8049, 8098,
8118, 8133, 8196, 9119, 9187, 9782,
10131, 10150, 10165, 10166, 10458,
10472, 11497, 11505, 11894, 11896,
13060, 13072, 13909, 13922, 18927,
18938, 18941, 18946, 18950, 18951,
18959, 18962, 18967, 18970, 19019,
19022, 19344, 19349, 21193, 21833,
21835, 21841, 22436, 22438, 24317
- \prg_return_true: 93,
94, 368, 369, 400, 492, 895, 897,
1842, 2056, 2073, 2081, 2086, 2099,
2104, 2390, 3421, 3432, 3443, 3453,
3467, 3485, 3498, 3513, 3808, 3829,
3845, 3863, 3876, 3894, 3905, 4103,
4110, 4116, 4124, 4132, 4657, 4665,
4690, 4918, 4943, 4961, 5521, 5553,
5593, 5605, 6208, 6214, 6240, 6274,
6306, 6362, 6370, 6379, 6385, 6386,
6483, 6485, 6487, 6489, 6609, 6612,
6618, 6765, 6779, 6883, 6913, 6923,
7441, 7446, 7451, 7456, 7463, 7470,
7475, 7480, 7485, 7490, 7495, 7500,

- 7505, 7510, 7526, 7532, 7540, 7589,
7642, 7668, 8037, 8047, 8098, 8120,
8131, 8196, 9122, 9185, 9190, 9192,
9783, 10131, 10166, 10457, 10473,
11496, 11504, 11887, 11892, 13055,
13078, 13911, 13920, 18916, 18930,
18938, 18941, 18946, 18950, 18962,
18967, 18970, 19017, 19340, 19346,
21191, 21833, 21835, 21841, 22435
- \prg_set_conditional:Nnn
..... 92, 1863, 6240
- \prg_set_conditional:Npnn 92, 93,
94, 1846, 2050, 2062, 2078, 2090, 6240
- \prg_set_eq_conditional:NNn
..... 93, 1959, 6240
- \prg_set_protected_conditional:Nnn
..... 92, 1863, 6240
- \prg_set_protected_conditional:Npnn
..... 92, 1846, 6240
- prg internal commands:
- __prg_break: 6496
- __prg_break:n 6496
- __prg_break_point: 6496
- __prg_break_point:Nn 325, 446, 6496
- __prg_generate_conditional:nnNnnnnn
..... 1858, 1883, 1892
- __prg_generate_conditional:nnnnnnnw
..... 1892
- __prg_generate_conditional_
count:nnNnn 1863
- __prg_generate_conditional_
count:nnNnnnn 1863
- __prg_generate_conditional_
parm:nnNpnn 1846
- __prg_generate_F_form:wnnnnnnn 1924
- __prg_generate_p_form:wnnnnnnn 1924
- __prg_generate_T_form:wnnnnnnn 1924
- __prg_generate_TF_form:wnnnnnnn 1924
- __prg_map_break:Nn 446, 6496
- __prg_replicate:N 6434
- __prg_replicate 6434
- __prg_replicate_0:n 6434
- __prg_replicate_1:n 6434
- __prg_replicate_2:n 6434
- __prg_replicate_3:n 6434
- __prg_replicate_4:n 6434
- __prg_replicate_5:n 6434
- __prg_replicate_6:n 6434
- __prg_replicate_7:n 6434
- __prg_replicate_8:n 6434
- __prg_replicate_9:n 6434
- __prg_replicate_first:N 6434
- __prg_replicate_first -:n 6434
- __prg_replicate_first_0:n 6434
- __prg_replicate_first_1:n ... 6434
- __prg_replicate_first_2:n ... 6434
- __prg_replicate_first_3:n ... 6434
- __prg_replicate_first_4:n ... 6434
- __prg_replicate_first_5:n ... 6434
- __prg_replicate_first_6:n ... 6434
- __prg_replicate_first_7:n ... 6434
- __prg_replicate_first_8:n ... 6434
- __prg_replicate_first_9:n ... 6434
- __prg_set_eq_conditional:NNNn 1959
- __prg_set_eq_conditional:nnNnnNNw
..... 1967, 1975
- __prg_set_eq_conditional_F_
form:nnn 1975
- __prg_set_eq_conditional_F_
form:wNnnnn 2020
- __prg_set_eq_conditional_
loop:nnnnNw 1975
- __prg_set_eq_conditional_p_
form:nnn 1975
- __prg_set_eq_conditional_p_
form:wNnnnn 2008
- __prg_set_eq_conditional_T_
form:nnn 1975
- __prg_set_eq_conditional_T_
form:wNnnnn 2016
- __prg_set_eq_conditional_TF_
form:nnn 1975
- __prg_set_eq_conditional_TF_
form:wNnnnn 2012
- \primitive 875
- prop commands:
- \c_empty_prop
133, 488, 7878, 7882, 7886, 7889, 8097
- \prop_clear:N
.. 128, 128, 7885, 7892, 20565, 22832
- \prop_clear_new:N
..... 128, 930, 7891, 22464, 22465
- \prop_const_from_keyval:Nn
..... 240, 7909, 22399, 22406
- \prop_count:N 240, 24083, 24112
- \prop_gclear:N 128, 7885, 7895
- \prop_gclear_new:N 128, 7891
- \prop_get:Nn 100, 27202, 27204
- \prop_get:NnN 60, 61, 129,
130, 7988, 23065, 23069, 23148, 23152
- \prop_get:NnNTF 129,
130, 131, 8126, 8560, 8580, 8635, 22591
- \prop_gpop:NnN 129, 7996
- \prop_gpop:NnNTF 129, 131, 8031
- \prop_gput:Nnn
... 129, 8053, 9104, 9150, 9271, 9304
- \prop_gput_if_new:Nnn 129, 8074
- \prop_gremove:Nn 130, 7972, 9158, 9313

- \prop_gset_eq:NN [128](#),
[7889](#), [7897](#), [22466](#), [22468](#), [22615](#), [22617](#)
 - \prop_gset_from_keyval:Nn . [240](#), [7909](#)
 - \prop_if_empty:NtF .. [130](#), [8095](#), [24109](#)
 - \prop_if_empty_p:N [130](#), [8095](#)
 - \prop_if_exist:NtF
..... [130](#), [7892](#), [7895](#), [8091](#)
 - \prop_if_exist_p:N [130](#), [8091](#)
 - \prop_if_in:NnTF [130](#), [8102](#)
 - \prop_if_in_p:Nn [130](#), [8102](#)
 - \prop_item:Nn
..... [130](#), [240](#), [8018](#), [27203](#), [27205](#)
 - \prop_log:N [133](#), [8172](#)
 - \prop_map_break:
... [132](#), [973](#), [8144](#), [8160](#), [8168](#), [24098](#)
 - \prop_map_break:n [132](#), [8168](#)
 - \prop_map_function:NN
..... [131](#), [239](#), [240](#), [494](#), [973](#),
[8137](#), [8182](#), [9172](#), [9327](#), [23223](#), [24088](#)
 - \prop_map_inline:Nn
..... [131](#), [8153](#), [21332](#),
[22910](#), [22929](#), [23117](#), [23126](#), [23718](#),
[23720](#), [23723](#), [23743](#), [23745](#), [23816](#),
[23833](#), [23877](#), [23879](#), [23883](#), [23886](#)
 - \prop_map_tokens:Nn [240](#), [24093](#)
 - \prop_new:N [128](#),
[128](#), [7879](#), [7892](#), [7895](#), [7905](#), [7906](#),
[7907](#), [7908](#), [8388](#), [8532](#), [9091](#), [9258](#),
[20499](#), [20500](#), [22991](#), [23032](#), [23708](#)
 - \prop_pop:NnN [129](#), [7996](#)
 - \prop_pop:NnNtF [129](#), [131](#), [8031](#)
 - \prop_put:Nnn [129](#), [338](#), [487](#),
[487](#), [8053](#), [8608](#), [8624](#), [8641](#), [20729](#),
[22644](#), [22652](#), [22655](#), [22661](#), [22664](#),
[22673](#), [22678](#), [22683](#), [22690](#), [22697](#),
[22934](#), [22992](#), [22994](#), [22996](#), [22998](#),
[23000](#), [23002](#), [23004](#), [23006](#), [23008](#),
[23010](#), [23012](#), [23014](#), [23016](#), [23018](#),
[23020](#), [23022](#), [23024](#), [23026](#), [23751](#),
[23753](#), [23759](#), [23761](#), [23770](#), [23776](#),
[23843](#), [23851](#), [23916](#), [23930](#), [23937](#)
 - \prop_put_if_new:Nnn [129](#), [8074](#)
 - \prop_rand_key_value:N ... [240](#), [24107](#)
 - \prop_remove:Nn [130](#),
[7972](#), [8605](#), [8620](#), [23112](#), [23115](#), [23119](#)
 - \prop_set_eq:NN
... [128](#), [7886](#), [7897](#), [20740](#), [22601](#),
[22603](#), [22608](#), [22610](#), [22866](#), [23107](#)
 - \prop_set_from_keyval:Nn .. [240](#), [7909](#)
 - \prop_show:N [132](#), [8172](#)
 - \g_tmpa_prop [133](#), [7905](#)
 - \l_tmpa_prop [133](#), [7905](#)
 - \g_tmpb_prop [133](#), [7905](#)
 - \l_tmpb_prop [133](#), [7905](#)
- prop internal commands:
- __prop_count:nn [24083](#)
 - __prop_from_keyval:n [7909](#)
 - __prop_from_keyval_key:n [7909](#)
 - __prop_from_keyval_key:w . [489](#), [7909](#)
 - __prop_from_keyval_loop:w ... [7909](#)
 - __prop_from_keyval_split:Nw . [7909](#)
 - __prop_from_keyval_value:n .. [7909](#)
 - __prop_from_keyval_value:w [489](#), [7909](#)
 - __prop_if_in:N [494](#), [8102](#)
 - __prop_if_in:nwn [494](#), [8102](#)
 - \l__prop_internal_tl .. [493](#), [7874](#),
[7877](#), [8057](#), [8063](#), [8064](#), [8080](#), [8087](#)
 - __prop_item:Nn:nwn [492](#)
 - __prop_item:Nn:nwn [8018](#)
 - __prop_map_function:Nwn [8137](#)
 - __prop_map_tokens:nwn [24093](#)
 - __prop_pair:wn [486](#), [487](#),
[487](#), [490](#), [494](#), [495](#), [496](#), [973](#), [7874](#),
[7875](#), [7951](#), [7966](#), [7969](#), [8021](#), [8024](#),
[8059](#), [8082](#), [8105](#), [8109](#), [8143](#), [8146](#),
[8156](#), [8158](#), [8163](#), [24097](#), [24100](#), [24117](#)
 - __prop_put:Nnn [8053](#)
 - __prop_put_if_new:Nnn [8074](#)
 - __prop_rand_item:w [24107](#)
 - __prop_show:NN [8172](#), [8174](#), [8176](#)
 - __prop_split:NnTF . [487](#), [493](#), [493](#),
[494](#), [7961](#), [7974](#), [7980](#), [7990](#), [7998](#),
[8007](#), [8033](#), [8043](#), [8062](#), [8085](#), [8128](#)
 - __prop_split_aux:NnTF [7961](#)
 - __prop_split_aux:w [490](#), [7961](#)
 - \protect [9433](#), [12538](#), [24744](#), [24771](#)
 - \protected [206](#), [208](#), [210](#), [235](#), [654](#), [7603](#), [7605](#)
 - \protrudechars [1011](#)
 - \ProvidesExplClass [6](#)
 - \ProvidesExplFile [6](#), [25675](#)
 - \ProvidesExplPackage [6](#)
 - pt [195](#)
- ptex commands:
- \ptex_autospacing:D [1189](#)
 - \ptex_autoxspacing:D [1190](#)
 - \ptex_dtou:D [1191](#)
 - \ptex_epTeXversion:D [1193](#)
 - \ptex_euc:D [1194](#)
 - \ptex_ifdbbox:D [1195](#)
 - \ptex_ifddir:D [1196](#)
 - \ptex_ifmdir:D [1197](#)
 - \ptex_iftbox:D [1198](#)
 - \ptex_iftdir:D [1199](#)
 - \ptex_ifybox:D [1200](#)
 - \ptex_ifydir:D [1201](#)
 - \ptex_inhibitglue:D [1202](#)
 - \ptex_inhibitxspcode:D [1203](#)
 - \ptex_inputencoding:D [1192](#)

`\ptex_jcharwidowpenalty:D` 1204
`\ptex_jfam:D` 1205
`\ptex_jfont:D` 1206
`\ptex_jis:D` 1207, 5393, 6551
`\ptex_kanjiskip:D` 1208, 6547
`\ptex_kansuji:D` 1209
`\ptex_kansujichar:D` 1210
`\ptex_kcatcode:D` 1211
`\ptex_kuten:D` 1212
`\ptex_noautospacing:D` 1213
`\ptex_noautoxspacing:D` 1214
`\ptex_postbreakpenalty:D` 1215
`\ptex_prebreakpenalty:D` 1216
`\ptex_ptexversion:D` 1217
`\ptex_showmode:D` 1218
`\ptex_sjis:D` 1219
`\ptex_tate:D` 1220
`\ptex_tbaselineshift:D` 1221
`\ptex_tfont:D` 1222
`\ptex_xkanjiskip:D` 1223
`\ptex_xspcode:D` 1224
`\ptex_ybaselineshift:D` 1225
`\ptex_yoko:D` 1226
`\ptexversion` 1217
`\pxdimen` 1012

Q

quark commands:

`\q_mark` 61, 104,
313, 337, 338, 353, 360, 363, 363,
374, 380, 387, 450, 454, 456, 457,
462, 463, 489, 490, 565, 796, 797,
798, 802, 2040, 2041, 2044, 2045,
2046, 2781, 2782, 2784, 2790, 2794,
2816, 2825, 2844, 2870, 2873, 2882,
2897, 2943, 2957, 2960, 2972, 2999,
3002, 3018, 3334, 3336, 3338, 3340,
3540, 3550, 3628, 3629, 3632, 3635,
3636, 3642, 3645, 3660, 3661, 3667,
3671, 3673, 3676, 4155, 4186, 4193,
4266, 4283, 4531, 4533, 4604, 5183,
5184, 5198, 5201, 5504, 5507, 5579,
5586, 6603, 6620, 6742, 6752, 6756,
6778, 6834, 6840, 6854, 6866, 6867,
6868, 6871, 6872, 6873, 6882, 6883,
6892, 7046, 7047, 7059, 7060, 7921,
7929, 7941, 7966, 7968, 7969, 8500,
8565, 8566, 8571, 8574, 9567, 9574,
9586, 10194, 10201, 10632, 10639,
10649, 10673, 10695, 10705, 10717,
11728, 11729, 11733, 17935, 17975,
17977, 17980, 17984, 17987, 17990,
17992, 17995, 25642, 25643, 25650

`\q_nil` 18, 18,
61, 61, 61, 297, 353, 356, 356, 363,
389, 391, 489, 1512, 1515, 3005,
3360, 3430, 3431, 3441, 3442, 3659,
3663, 3681, 3684, 3687, 3777, 3778,
4604, 4656, 4675, 4677, 4688, 4689,
7934, 7938, 7945, 9454, 9461, 10649,
10659, 10673, 10683, 19813, 19831
`\q_no_value` 60,
61, 61, 61, 66, 67, 67, 67, 67, 72,
72, 72, 111, 129, 129, 129, 389, 391,
401, 402, 454, 491, 491, 4604, 4664,
4681, 4683, 4949, 4957, 4969, 4995,
6725, 6740, 7992, 8003, 8012, 9979
`\quark_if_nil:n` 391
`\quark_if_nil:NTF`
. 61, 4654, 19835, 19855
`\quark_if_nil:nTF` . 61, 390, 3358, 4672
`\quark_if_nil_p:N` 61, 4654
`\quark_if_nil_p:n` 61, 4672
`\quark_if_no_value:NTF`
. 61, 4654, 23067, 23071, 23150, 23154
`\quark_if_no_value:nTF` 61, 4672
`\quark_if_no_value_p:N` 61, 4654
`\quark_if_no_value_p:n` 61, 4672
`\quark_if_recursion_tail_break:N`
. 27206
`\quark_if_recursion_tail_break:n`
. 27208
`\quark_if_recursion_tail_-`
break:NN 62, 3591, 4235, 4253, 4642
`\quark_if_recursion_tail_-`
break:nN
. 62, 3562, 3925, 4642, 6941, 6954
`\quark_if_recursion_tail_stop:N` .
. . 62, 4610, 6082, 6999, 23566, 25292
`\quark_if_recursion_tail_stop:n` .
. 62,
371, 390, 4624, 6593, 7029, 7926, 25125
`\quark_if_recursion_tail_stop_-`
do:Nn 62, 4570, 4610, 6021,
6038, 6085, 24557, 24565, 24732, 24752
`\quark_if_recursion_tail_stop_-`
do:nn . . 62, 4624, 6362, 6377, 24804
`\quark_new:N` 60, 4598,
4604, 4605, 4606, 4607, 4608, 4609
`\q_recursion_stop` 18, 18, 61,
62, 62, 62, 62, 63, 297, 309, 389,
461, 1514, 1518, 1904, 1972, 2768,
2998, 4547, 4555, 4560, 4608, 6016,
6033, 6076, 6103, 6359, 6374, 6589,
6987, 7023, 7922, 19813, 19831,
23574, 24497, 24500, 24509, 24520,
24534, 24548, 24555, 24561, 24580,

- 24582, 24591, 24593, 24600, 24601,
 24604, 24607, 24620, 24726, 24748,
 24770, 24800, 24816, 24818, 24821,
 24824, 24826, 24831, 24834, 24836,
 24839, 24842, 24847, 24870, 24873,
 24875, 24877, 24882, 24963, 24967,
 24969, 24974, 24988, 25013, 25017,
 25019, 25024, 25032, 25264, 25312,
 25329, 25333, 25335, 25340, 25355
 \q_recursion_tail 61, 62,
 62, 62, 62, 62, 63, 309, 379, 389,
 390, 459, 494, 1904, 1909, 1972,
 1991, 3555, 3573, 3583, 3914, 4200,
 4209, 4226, 4246, 4547, 4608, 4612,
 4618, 4627, 4634, 4639, 4644, 4651,
 6016, 6033, 6076, 6359, 6374, 6589,
 6935, 6949, 6969, 6987, 7023, 7922,
 8106, 8117, 10632, 10639, 10700,
 23574, 24497, 24561, 24595, 24726,
 24748, 24779, 24800, 25263, 25311
 \q_stop 18, 18, 29, 44, 60,
 60, 61, 61, 104, 297, 313, 338, 353,
 362, 380, 384, 389, 416, 428, 457,
 462, 477, 479, 489, 490, 518, 596,
 796, 797, 798, 1513, 1516, 1652,
 1654, 1920, 1925, 1942, 1948, 1954,
 2002, 2008, 2012, 2016, 2020, 2041,
 2044, 2045, 2046, 2785, 2794, 2820,
 2870, 2874, 2878, 2886, 2892, 2901,
 2907, 2909, 2943, 2963, 2972, 2999,
 3359, 3512, 3518, 3540, 3550, 3630,
 3632, 3637, 3639, 3665, 3687, 3768,
 3770, 3787, 3805, 3822, 3844, 3945,
 3955, 4052, 4155, 4186, 4193, 4266,
 4275, 4281, 4283, 4289, 4306, 4325,
 4387, 4444, 4456, 4494, 4510, 4517,
 4525, 4527, 4531, 4533, 4604, 4969,
 4972, 4980, 4982, 5063, 5064, 5185,
 5198, 5201, 5203, 5483, 5499, 5501,
 5505, 5518, 5579, 5586, 6009, 6015,
 6032, 6727, 6730, 6742, 6745, 6753,
 6756, 6764, 6778, 6840, 6868, 6871,
 6872, 6884, 6892, 7048, 7059, 7060,
 7061, 7087, 7121, 7519, 7522, 7552,
 7586, 7623, 7627, 7633, 7656, 7840,
 7847, 7856, 7865, 7928, 7938, 7941,
 7945, 7966, 7969, 8502, 8506, 8508,
 8567, 9454, 9495, 9552, 9590, 9603,
 9649, 9651, 9675, 9680, 9847, 9850,
 9864, 9879, 9881, 10141, 10165,
 10194, 10201, 10473, 10475, 10703,
 10720, 10747, 10766, 10767, 10833,
 10835, 10855, 10859, 10868, 10876,
 10887, 11334, 11342, 11352, 11480,
 11482, 11487, 11705, 11716, 11723,
 11729, 11733, 11747, 11766, 12578,
 12582, 13089, 13094, 13665, 13687,
 13907, 13908, 13933, 13934, 14100,
 14101, 14102, 14269, 14270, 17377,
 17442, 17923, 17932, 17936, 17938,
 17975, 17976, 17977, 17982, 17984,
 17988, 17990, 17998, 23268, 23270,
 23271, 23272, 23274, 23276, 23278,
 23467, 23482, 23486, 23500, 23503,
 23518, 23524, 23526, 23527, 23529,
 23532, 23556, 24050, 24113, 24126,
 24127, 24129, 24133, 24137, 24139,
 24144, 25631, 25644, 25653, 25720,
 25723, 25769, 25772, 26871, 26875
 \s_stop 63, 64, 797, 4710, 4711, 15870,
 15885, 17056, 17060, 17936, 17938
 quark internal commands:
 \s__fp 590, 592, 596, 597, 620,
 626, 628, 630, 644, 646, 647, 677,
 680, 681, 682, 684, 690, 692, 776,
 11588, 11598, 11599, 11600, 11601,
 11602, 11612, 11617, 11619, 11620,
 11641, 11644, 11646, 11656, 11668,
 11688, 11705, 11708, 11715, 11722,
 11738, 11765, 11871, 11873, 11875,
 11876, 11877, 11879, 11880, 11881,
 11883, 11899, 12084, 12089, 12316,
 12362, 12371, 12373, 13050, 13208,
 13665, 13680, 13704, 13724, 13725,
 13908, 13933, 13934, 13948, 13949,
 13986, 13987, 14100, 14101, 14102,
 14111, 14127, 14131, 14195, 14196,
 14199, 14210, 14211, 14219, 14220,
 14222, 14223, 14224, 14226, 14227,
 14228, 14240, 14243, 14247, 14250,
 14270, 14318, 14321, 14324, 14344,
 14345, 14347, 14348, 14349, 14357,
 14360, 14371, 14372, 14374, 14383,
 14459, 14611, 14645, 14646, 14649,
 14730, 14868, 14876, 14878, 15055,
 15063, 15065, 15067, 15070, 15571,
 15583, 15585, 15794, 15811, 15813,
 15999, 16018, 16020, 16021, 16024,
 16041, 16044, 16047, 16072, 16073,
 16075, 16091, 16176, 16189, 16191,
 16194, 16199, 16261, 16274, 16276,
 16289, 16291, 16304, 16306, 16319,
 16321, 16334, 16336, 16349, 16359,
 16780, 16796, 16797, 16801, 16812,
 16919, 16932, 16934, 16950, 16953,
 16963, 16986, 16997, 16999, 17013,
 17015, 17020, 17082, 17103, 17106,
 17136, 17157, 17160, 17210, 17226,

- 17229, 17305, 17306, 17436, 17438
 \s__fp_⟨type⟩ 620
 \s__fp_division 11593
 \s__fp_exact 11593, 11598,
 11599, 11600, 11601, 11602, 14195
 \s__fp_invalid 11593
 \s__fp_mark ... 605, 626, 627, 630,
 652, 655, 664, 11591, 11940, 11941,
 11943, 11947, 13257, 13270, 13352,
 13782, 13783, 13786, 13789, 13790
 \s__fp_overflow 11593, 11619
 \s__fp_stop
 . 598, 11591, 11779, 13159, 13258,
 13262, 13271, 13784, 13789, 13790,
 13792, 14277, 14288, 14298, 14306
 \s__fp_tuple 595,
 11689, 11695, 11696, 11773, 11775,
 13447, 13657, 13672, 13697, 13699,
 13716, 13717, 13719, 13978, 13979,
 15101, 15102, 15108, 15109, 17032
 \s__fp_underflow 11593, 11617
 \s__prop 486, 487, 490,
 496, 973, 974, 7874, 7874, 7875,
 7878, 7920, 7952, 7966, 7969, 8021,
 8024, 8060, 8083, 8105, 8109, 8143,
 8146, 8158, 24097, 24100, 24117, 24122
 __quark_if_empty_return:n ... 4672
 __quark_if_nil:w 391, 4672
 __quark_if_no_value:w 4672
 __quark_if_recursion_tail:w ...
 390, 4624, 4651
 \s__seq ... 393, 396, 397, 402, 407,
 974, 4715, 4724, 4754, 4759, 4764,
 4769, 4802, 4828, 4836, 4840, 5016,
 5064, 5195, 24127, 24133, 24174, 24190
 \s__tl 801, 802, 802, 803,
 803, 811, 811, 812, 18064, 18065,
 18284, 18315, 18321, 18346, 18364,
 18369, 18383, 18395, 18418, 18421
 \q__tl_act_mark
 364, 364, 365, 3691, 3696, 3713
 \q__tl_act_stop
 364, 3691, 3696, 3700, 3709,
 3711, 3717, 3722, 3725, 3729, 3732
 \quitvmode 792
- ## R
- \r 25318
 \radical 516
 \raise 517
 rand 195
 randint 195
 \randomseed 1013
 \read 518
 \readline 655
 \ref 25366
 regex commands:
 \c_foo_regex 202
 \regex(g)set:Nn 209
 \regex_const:Nn 202, 209, 21114
 \regex_count:NnN 210, 21157
 \regex_count:nnN 210, 896, 21157
 \regex_extract_all:NnN ... 210, 21161
 \regex_extract_all:nnN
 203, 210, 818, 21161
 \regex_extract_all:NnNTF . 210, 21161
 \regex_extract_all:nnNTF . 210, 21161
 \regex_extract_once:NnN .. 210, 21161
 \regex_extract_once:nnN
 210, 210, 21161
 \regex_extract_once:NnNTF 210, 21161
 \regex_extract_once:nnNTF
 207, 210, 21161
 \regex_gset:Nn 209, 21114
 \regex_match:NnTF 209, 21147
 \regex_match:nnTF 209, 21147
 \regex_new:N 209,
 820, 21108, 21110, 21111, 21112, 21113
 \regex_replace_all:NnN ... 211, 21161
 \regex_replace_all:nnN 203, 211, 21161
 \regex_replace_all:NnNTF . 211, 21161
 \regex_replace_all:nnNTF . 211, 21161
 \regex_replace_once:NnN .. 211, 21161
 \regex_replace_once:nnN
 210, 211, 21161
 \regex_replace_once:NnNTF 211, 21161
 \regex_replace_once:nnNTF 211, 21161
 \regex_set:Nn 209, 21114
 \regex_show:N 209, 21129
 \regex_show:n 202, 209, 21129
 \regex_split:NnN 211, 21161
 \regex_split:nnN 211, 21161
 \regex_split:NnNTF 211, 21161
 \regex_split:nnNTF 211, 21161
 \g_tmpa_regex 211, 21110
 \l_tmpa_regex 211, 21110
 \g_tmpb_regex 211, 21110
 \l_tmpb_regex 211, 21110
 regex internal commands:
 __regex_action_cost:n . 862, 865,
 874, 20223, 20224, 20232, 20679, 20705
 __regex_action_free:n
 212, 862, 873, 20246, 20252,
 20253, 20264, 20329, 20333, 20358,
 20383, 20387, 20390, 20418, 20426,
 20436, 20450, 20481, 20677, 20681
 __regex_action_free_aux:nn .. 20681

- _regex_action_free_group:n [212](#),
[862](#), [873](#), [20276](#), [20398](#), [20401](#), [20681](#)
- _regex_action_start_wildcard: .
..... [862](#), [20140](#), [20674](#)
- _regex_action_submatch:n
[863](#), [20352](#), [20353](#), [20479](#), [20725](#), [20727](#)
- _regex_action_success:
..... [212](#), [862](#), [20143](#), [20162](#), [20732](#)
- _regex_action_wildcard: [878](#)
- \c_regex_all_catcodes_int
..... [18985](#), [19095](#), [19185](#), [19785](#)
- _regex_anchor:N
..... [832](#), [872](#), [19416](#), [19980](#), [20441](#)
- \c_regex_ascii_lower_int
..... [18572](#), [18630](#), [18636](#)
- \c_regex_ascii_max_control_int .
..... [18569](#), [18746](#)
- \c_regex_ascii_max_int
..... [18569](#), [18739](#), [18747](#), [18937](#)
- \c_regex_ascii_min_int
..... [18569](#), [18738](#), [18745](#)
- _regex_assertion:Nn [832](#),
[872](#), [19416](#), [19441](#), [19450](#), [19974](#), [20441](#)
- _regex_b_test:
[832](#), [872](#), [19441](#), [19450](#), [19979](#), [20441](#)
- \l_regex_balance_int
.. [820](#), [885](#), [18567](#), [20521](#), [20630](#),
[20634](#), [20635](#), [20823](#), [20852](#), [21051](#),
[21068](#), [21363](#), [21389](#), [21412](#), [21416](#),
[21417](#), [21424](#), [21425](#), [21429](#), [21430](#)
- \g_regex_balance_intarray
[818](#), [820](#), [18564](#), [20629](#), [20793](#), [20808](#)
- \l_regex_balance_tl [212](#),
[885](#), [887](#), [20748](#), [20824](#), [20853](#), [20915](#)
- _regex_branch:n
.... [832](#), [849](#), [868](#), [18561](#), [19100](#),
[19596](#), [19649](#), [19827](#), [19956](#), [20321](#)
- _regex_break_point:TF
..... [821](#), [844](#), [865](#), [18573](#),
[18579](#), [20223](#), [20224](#), [20447](#), [20470](#)
- _regex_break_true:w
.. [821](#), [821](#), [18573](#), [18579](#), [18584](#),
[18591](#), [18598](#), [18604](#), [18611](#), [18619](#),
[18666](#), [18678](#), [18695](#), [19391](#), [20462](#)
- _regex_build:N
[895](#), [20121](#), [21154](#), [21160](#), [21164](#), [21168](#)
- _regex_build:n
[895](#), [20121](#), [21149](#), [21158](#), [21163](#), [21166](#)
- _regex_build_for_cs:n [18690](#), [20145](#)
- _regex_build_new_state:
..... [20137](#), [20138](#),
[20154](#), [20155](#), [20184](#), [20205](#), [20237](#),
[20274](#), [20279](#), [20326](#), [20341](#), [20346](#),
[20385](#), [20404](#), [20439](#), [20443](#), [20476](#)
- \l_regex_build_tl [849](#),
[18558](#), [19092](#), [19099](#), [19117](#), [19122](#),
[19125](#), [19126](#), [19129](#), [19130](#), [19133](#),
[19179](#), [19182](#), [19215](#), [19229](#), [19233](#),
[19360](#), [19374](#), [19415](#), [19440](#), [19449](#),
[19459](#), [19491](#), [19504](#), [19508](#), [19589](#),
[19592](#), [19595](#), [19601](#), [19602](#), [19605](#),
[19648](#), [19915](#), [19931](#), [19949](#), [19955](#),
[20010](#), [20013](#), [20018](#), [20048](#), [20063](#),
[20067](#), [20070](#), [20076](#), [20822](#), [20841](#),
[20856](#), [20859](#), [20880](#), [20912](#), [20975](#),
[20978](#), [20994](#), [21042](#), [21058](#), [21094](#)
- _regex_build_transition_
left:NNN [20180](#), [20387](#), [20401](#), [20418](#)
- _regex_build_transition_
right:nNn [20180](#),
[20238](#), [20276](#), [20329](#), [20333](#), [20358](#),
[20383](#), [20390](#), [20398](#), [20426](#), [20436](#)
- _regex_build_transitions_
lazyness:NNNN
..... [20203](#), [20245](#), [20251](#), [20263](#)
- \l_regex_capturing_group_int ...
[818](#), [862](#), [902](#), [20120](#), [20135](#), [20292](#),
[20294](#), [20305](#), [20306](#), [20314](#), [20315](#),
[20318](#), [20911](#), [21322](#), [21394](#), [21402](#)
- \l_regex_case_changed_char_int .
..... [822](#),
[18600](#), [18603](#), [18614](#), [18617](#), [18618](#),
[18625](#), [18629](#), [18635](#), [20494](#), [20599](#)
- \c_regex_catcode_A_int [18985](#)
- \c_regex_catcode_B_int [18985](#)
- \c_regex_catcode_C_int [18985](#)
- \c_regex_catcode_D_int [18985](#)
- \c_regex_catcode_E_int [18985](#)
- \c_regex_catcode_in_class_mode_
int [18975](#),
[19084](#), [19458](#), [19618](#), [19710](#), [19739](#)
- \g_regex_catcode_intarray
..... [818](#), [820](#), [18564](#), [20627](#), [20644](#)
- \c_regex_catcode_L_int [18985](#)
- \c_regex_catcode_M_int [18985](#)
- \c_regex_catcode_mode_int [18975](#),
[19080](#), [19153](#), [19490](#), [19708](#), [19737](#)
- \c_regex_catcode_O_int [18985](#)
- \c_regex_catcode_P_int [18985](#)
- \c_regex_catcode_S_int [18985](#)
- \c_regex_catcode_T_int [18985](#)
- \c_regex_catcode_U_int [18985](#)
- \l_regex_catcodes_bool
..... [18982](#), [19744](#), [19748](#), [19783](#)
- \l_regex_catcodes_int [833](#),
[18982](#), [19096](#), [19184](#), [19186](#), [19192](#),
[19477](#), [19494](#), [19593](#), [19606](#), [19704](#),
[19741](#), [19776](#), [19778](#), [19784](#), [19785](#)

- _regex_char_if_alphanumeric:N .
..... 18955
- _regex_char_if_alphanumeric:NTF
..... 18933, 19146, 21025
- _regex_char_if_special:N ... 18933
- _regex_char_if_special:NTF ...
..... 18933, 19142
- _g_regex_charcode_intarray
..... 818, 820, 18564, 20625, 20641
- _regex_chk_c_allowed:TF
..... 19065, 19697
- _regex_class:NnnnN
..... 831, 839, 840, 846,
18562, 19180, 19485, 19486, 19492,
19843, 19923, 19933, 19971, 20217
- _c_regex_class_mode_int
..... 18975, 19070, 19085
- _regex_class_repeat:n
.... 866, 20227, 20233, 20249, 20258
- _regex_class_repeat:nN 20228, 20242
- _regex_class_repeat:nnN
..... 20229, 20256
- _regex_command_K:
..... 832, 19949, 19972, 20474
- _regex_compile:n 19135,
20123, 21116, 21121, 21126, 21131
- _regex_compile:w
..... 838, 19089, 19137, 19790
- _regex_compile_\$: 19411
- _regex_compile(: 19613
- _regex_compile): 19652
- _regex_compile.: 19382
- _regex_compile_/A: 19411
- _regex_compile_/B: 19435
- _regex_compile_/b: 19435
- _regex_compile_/c: 19696
- _regex_compile_/D: 19394
- _regex_compile_/d: 19394
- _regex_compile_/G: 19411
- _regex_compile_/H: 19394
- _regex_compile_/h: 19394
- _regex_compile_/K: 19946
- _regex_compile_/N: 19394
- _regex_compile_/S: 19394
- _regex_compile_/s: 19394
- _regex_compile_/u: 19863
- _regex_compile_/V: 19394
- _regex_compile_/v: 19394
- _regex_compile_/W: 19394
- _regex_compile_/w: 19394
- _regex_compile_/Z: 19411
- _regex_compile_/z: 19411
- _regex_compile[: 19469
- _regex_compile]: 19453
- _regex_compile^: 19411
- _regex_compile_abort_tokens:n .
..... 19195, 19222, 19573, 19583
- _regex_compile_anchor:NTF .. 19411
- _regex_compile_c[:w 19733
- _regex_compile_c_C:NN 19712, 19721
- _regex_compile_c_lbrack_add:N .
..... 19733
- _regex_compile_c_lbrack_end: 19733
- _regex_compile_c_lbrack_-
loop:NN 19733
- _regex_compile_c_test:NN ... 19696
- _regex_compile_class:NN 19499
- _regex_compile_class:TFNN
..... 847, 19484, 19495, 19499
- _regex_compile_class_catcode:w
..... 19476, 19488
- _regex_compile_class_normal:w .
..... 19479, 19482
- _regex_compile_class_posix:NNNNw
..... 19518
- _regex_compile_class_posix_-
end:w 19518
- _regex_compile_class_posix_-
loop:w 19518
- _regex_compile_class_posix_-
test:w 19472, 19518
- _regex_compile_cs_aux:Nn ... 19799
- _regex_compile_cs_aux:NNnnnN 19799
- _regex_compile_end:
..... 838, 19089, 19162, 19808
- _regex_compile_end_cs: 19158, 19799
- _regex_compile_escaped:N
..... 19147, 19164
- _regex_compile_group_begin:N ..
.. 19587, 19635, 19640, 19658, 19660
- _regex_compile_group_end: ...
..... 19587, 19655
- _regex_compile_lparen:w
..... 19622, 19626
- _regex_compile_one:n
.... 19174, 19326, 19332, 19386,
19397, 19400, 19410, 19564, 19815
- _regex_compile_quantifier:w ...
..... 19193, 19204, 19464, 19607
- _regex_compile_quantifier_*:w .
..... 19238
- _regex_compile_quantifier_+:w .
..... 19238
- _regex_compile_quantifier_?:w .
..... 19238
- _regex_compile_quantifier_-
abort:nNN
.. 19213, 19248, 19267, 19281, 19305

__regex_compile_quantifier_
 braced_auxi:w [19244](#)
 __regex_compile_quantifier_
 braced_auxii:w [19244](#)
 __regex_compile_quantifier_
 braced_auxiii:w [19244](#)
 __regex_compile_quantifier_
 lazyness:nnNN [841](#), [19225](#), [19239](#),
 [19241](#), [19243](#), [19256](#), [19277](#), [19300](#)
 __regex_compile_quantifier_
 none: [19209](#), [19211](#), [19213](#)
 __regex_compile_range:Nw
 [19324](#), [19337](#)
 __regex_compile_raw:N
 [19015](#), [19143](#), [19147](#), [19149](#),
 [19167](#), [19172](#), [19200](#), [19317](#), [19319](#),
 [19339](#), [19385](#), [19431](#), [19467](#), [19515](#),
 [19535](#), [19553](#), [19610](#), [19615](#), [19620](#),
 [19636](#), [19646](#), [19654](#), [19672](#), [19673](#),
 [19674](#), [19681](#), [19690](#), [19691](#), [19692](#),
 [19700](#), [19755](#), [19804](#), [19876](#), [19882](#)
 __regex_compile_raw_error:N ...
 [19314](#),
 [19422](#), [19438](#), [19447](#), [19866](#), [19950](#)
 __regex_compile_special:N [19143](#),
 [19164](#), [19206](#), [19227](#), [19254](#), [19259](#),
 [19275](#), [19289](#), [19323](#), [19342](#), [19502](#),
 [19520](#), [19539](#), [19560](#), [19628](#), [19663](#),
 [19681](#), [19723](#), [19742](#), [19869](#), [19885](#)
 __regex_compile_special_group_
 -:w [19661](#)
 __regex_compile_special_group_
 ::w [19657](#)
 __regex_compile_special_group_
 i:w [19661](#)
 __regex_compile_special_group_
 l:w [19657](#)
 __regex_compile_u_end:
 [19888](#), [19894](#), [19899](#)
 __regex_compile_u_in_cs:
 [19905](#), [19908](#)
 __regex_compile_u_in_cs_aux:n ..
 [19918](#), [19921](#)
 __regex_compile_u_loop:NN ... [19863](#)
 __regex_compile_u_not_cs:
 [19903](#), [19927](#)
 __regex_compile_|: [19644](#)
 __regex_compute_case_changed_
 char: [18601](#), [18615](#), [18623](#)
 __regex_count:nnN [21158](#), [21160](#), [21205](#)
 \c__regex_cs_in_class_mode_int ..
 [18975](#), [19796](#)
 \c__regex_cs_mode_int . [18975](#), [19794](#)
 \l__regex_cs_name_tl
 [18568](#), [18687](#), [18693](#)
 \l__regex_curr_catcode_int [18645](#),
 [18664](#), [18672](#), [18684](#), [20494](#), [20643](#)
 \l__regex_curr_char_int
 [18583](#), [18589](#),
 [18590](#), [18597](#), [18609](#), [18610](#), [18625](#),
 [18626](#), [18627](#), [18628](#), [18634](#), [18665](#),
 [19390](#), [20468](#), [20494](#), [20598](#), [20640](#)
 __regex_curr_cs_to_str:
 [18544](#), [18675](#), [18687](#)
 \l__regex_curr_pos_int [819](#), [18547](#),
 [20161](#), [20461](#), [20489](#), [20522](#), [20524](#),
 [20527](#), [20558](#), [20568](#), [20597](#), [20612](#),
 [20626](#), [20628](#), [20630](#), [20631](#), [20632](#),
 [20642](#), [20645](#), [20730](#), [20739](#), [21257](#)
 \l__regex_curr_state_int
 [874](#), [880](#), [20498](#), [20650](#),
 [20656](#), [20657](#), [20659](#), [20664](#), [20667](#),
 [20689](#), [20694](#), [20699](#), [20700](#), [20708](#)
 \l__regex_curr_submatches_prop ..
 [20499](#), [20565](#), [20669](#),
 [20701](#), [20702](#), [20720](#), [20729](#), [20741](#)
 \l__regex_default_catcodes_int ..
 [833](#), [18982](#), [19094](#),
 [19096](#), [19192](#), [19494](#), [19593](#), [19606](#)
 __regex_disable_submatches: ...
 .. [18689](#), [19791](#), [20722](#), [21199](#), [21208](#)
 \l__regex_empty_success_bool ...
 .. [20507](#), [20550](#), [20554](#), [20737](#), [21267](#)
 __regex_escape_\u:w [18821](#)
 __regex_escape_/a:w [18821](#)
 __regex_escape_/break:w [18821](#)
 __regex_escape_/e:w [18821](#)
 __regex_escape_/f:w [18821](#)
 __regex_escape_/n:w [18821](#)
 __regex_escape_/r:w [18821](#)
 __regex_escape_/t:w [18821](#)
 __regex_escape_/x:w [18840](#)
 __regex_escape_\w [18805](#)
 __regex_escape_break:w [18821](#)
 __regex_escape_escaped:N
 [18791](#), [18815](#), [18818](#)
 __regex_escape_loop:N
 [827](#), [18798](#), [18805](#), [18840](#),
 [18876](#), [18884](#), [18885](#), [18902](#), [18911](#)
 __regex_escape_raw:N
 . [828](#), [18792](#), [18818](#), [18829](#), [18831](#),
 [18833](#), [18835](#), [18837](#), [18839](#), [18853](#)
 __regex_escape_unescaped:N
 [18790](#), [18808](#), [18818](#)
 __regex_escape_use:nnnn
 [826](#), [838](#), [18777](#), [19140](#), [20825](#)
 __regex_escape_x:N [828](#), [18875](#), [18879](#)

_regex_escape_x_end:w .. [828](#), [18840](#)
 _regex_escape_x_large:n [18840](#)
 _regex_escape_x_loop:N
 [828](#), [18872](#), [18888](#)
 _regex_escape_x_loop_error: . [18888](#)
 _regex_escape_x_loop_error:n ..
 [18891](#), [18903](#), [18908](#)
 _regex_escape_x_test:N
 [828](#), [18843](#), [18857](#)
 _regex_escape_x_testii:N ... [18857](#)
 \l_regex_every_match_tl
 [20506](#), [20572](#), [20576](#), [20585](#)
 _regex_extract: [897](#), [21223](#),
 [21229](#), [21241](#), [21318](#), [21362](#), [21385](#)
 _regex_extract_all:nnN [21172](#), [21217](#)
 _regex_extract_b:wn [21318](#)
 _regex_extract_e:wn [21318](#)
 _regex_extract_once:nnN
 [21170](#), [21217](#)
 _regex_extract_seq_aux:n
 [21283](#), [21301](#)
 _regex_extract_seq_aux:ww .. [21301](#)
 \l_regex_fresh_thread_bool
 [875](#), [880](#), [20480](#), [20486](#),
 [20507](#), [20610](#), [20676](#), [20678](#), [20738](#)
 _regex_get_digits:NTFw
 [19001](#), [19246](#), [19261](#)
 _regex_get_digits_loop:nw
 [19004](#), [19007](#), [19010](#)
 _regex_get_digits_loop:w ... [19001](#)
 _regex_group:nnnN [832](#),
 [849](#), [19635](#), [19640](#), [19965](#), [20141](#), [20289](#)
 _regex_group_aux:nnnnN
 [868](#), [20268](#), [20291](#), [20299](#), [20302](#)
 _regex_group_aux:nnnnnN [868](#)
 _regex_group_end_extract_seq:N
 [21224](#), [21232](#), [21272](#), [21274](#)
 _regex_group_end_replace:N ...
 [21379](#), [21408](#), [21410](#)
 \l_regex_group_level_int
 [18974](#), [19093](#),
 [19111](#), [19113](#), [19115](#), [19594](#), [19600](#)
 _regex_group_no_capture:nnnN ..
 [832](#), [19658](#), [19967](#), [20289](#)
 _regex_group_repeat:nn [20284](#), [20336](#)
 _regex_group_repeat:nnN
 [20285](#), [20376](#)
 _regex_group_repeat:nnnN
 [20286](#), [20407](#)
 _regex_group_repeat_aux:n
 [869](#), [870](#), [20343](#), [20356](#), [20394](#), [20411](#)
 _regex_group_resetting:nnnN ...
 [832](#), [19660](#), [19969](#), [20300](#)
 _regex_group_resetting_
 loop:nnNn [20300](#)
 _regex_group_submatches:nnN ...
 .. [20344](#), [20349](#), [20379](#), [20395](#), [20409](#)
 _regex_hexadecimal_use:N ... [18913](#)
 _regex_hexadecimal_use:NTF ...
 [18874](#), [18883](#), [18893](#), [18913](#)
 _regex_if_end_range:NN [19337](#)
 _regex_if_end_range:NNTF ... [19337](#)
 _regex_if_in_class:TF
 [19025](#), [19104](#), [19177](#),
 [19193](#), [19321](#), [19384](#), [19455](#), [19471](#),
 [19615](#), [19646](#), [19654](#), [21485](#), [21498](#)
 _regex_if_in_class_or_catcode:TF
 .. [19045](#), [19413](#), [19437](#), [19446](#), [19865](#)
 _regex_if_in_cs:TF
 [19033](#), [19802](#), [21483](#), [21492](#)
 _regex_if_match:nn
 [21149](#), [21154](#), [21196](#)
 _regex_if_raw_digit:NN [19013](#)
 _regex_if_raw_digit:NNTF
 [19003](#), [19009](#), [19013](#)
 _regex_if_two_empty_matches:TF
 [875](#), [20507](#), [20555](#), [20561](#), [20734](#)
 _regex_if_within_catcode:TF ...
 [19057](#), [19474](#)
 _regex_int_eval:w
 .. [18506](#), [20761](#), [20762](#), [20773](#), [21339](#)
 \l_regex_internal_a_int
 [841](#), [888](#), [18550](#),
 [19246](#), [19257](#), [19268](#), [19278](#), [19282](#),
 [19291](#), [19294](#), [19298](#), [19301](#), [19308](#),
 [20250](#), [20253](#), [20259](#), [20264](#), [20345](#),
 [20360](#), [20366](#), [20372](#), [20381](#), [20384](#),
 [20388](#), [20391](#), [20396](#), [20399](#), [20402](#),
 [20417](#), [20425](#), [20434](#), [20928](#), [20949](#)
 \l_regex_internal_a_tl
 [826](#), [856](#), [857](#), [858](#), [899](#), [902](#),
 [18550](#), [18674](#), [18677](#), [18781](#), [18789](#),
 [18796](#), [18803](#), [19542](#), [19547](#), [19562](#),
 [19567](#), [19572](#), [19576](#), [19582](#), [19583](#),
 [19810](#), [19821](#), [19871](#), [19901](#), [19913](#),
 [19929](#), [19959](#), [19962](#), [20013](#), [20028](#),
 [20070](#), [20077](#), [20175](#), [20176](#), [20177](#),
 [20178](#), [20327](#), [20328](#), [20332](#), [20334](#),
 [21135](#), [21144](#), [21368](#), [21398](#), [21428](#)
 \l_regex_internal_b_int
 [18550](#), [19261](#), [19292](#), [19295](#),
 [19296](#), [19298](#), [19302](#), [19309](#), [20361](#),
 [20366](#), [20371](#), [20417](#), [20425](#), [20434](#)
 \l_regex_internal_b_tl [18550](#)
 \l_regex_internal_bool
 .. [18550](#), [19541](#), [19546](#), [19566](#), [19575](#)

```

\l_regex_internal_c_int .....
.. 18550, 20363, 20368, 20369, 20373
\l_regex_internal_regex .....
. 837, 18998, 19133, 19812, 19818,
  20124, 21117, 21122, 21127, 21132
\l_regex_internal_seq ... 18550,
  20094, 20095, 20100, 20107, 20108,
  20109, 20111, 21278, 21296, 21299
\g_regex_internal_tl .....
.. 18550, 18794, 18798, 19910, 19917
\__regex_item_caseful_equal:n ...
..... 832, 18581, 18706,
  18707, 18711, 18712, 18713, 18714,
  18715, 18724, 18729, 18747, 18765,
  19097, 19684, 19845, 19924, 19981
\__regex_item_caseful_range:nn ..
..... 832, 18581, 18703,
  18718, 18721, 18722, 18723, 18737,
  18744, 18751, 18753, 18755, 18758,
  18759, 18760, 18761, 18766, 18769,
  18774, 18775, 19098, 19686, 19983
\__regex_item_caseless_equal:n ..
..... 832, 18595, 19666, 19988
\__regex_item_caseless_range:nn .
..... 832, 18595, 19668, 19990
\__regex_item_catcode: ..... 18642
\__regex_item_catcode:nTF .....
  832, 847, 18642, 19186, 19496, 19995
\__regex_item_catcode_reverse:nTF
..... 832, 18642, 19497, 19997
\__regex_item_cs:n .....
..... 821, 832, 18682, 19818, 20004
\__regex_item_equal:n .....
.... 18640, 19097, 19327, 19333,
  19363, 19376, 19377, 19665, 19683
\__regex_item_exact:nn .....
..... 832, 857, 18662, 19939, 20001
\__regex_item_exact_cs:n .....
  832, 854, 18662, 19820, 19936, 20003
\__regex_item_range:nn .....
.. 18640, 19098, 19365, 19667, 19685
\__regex_item_reverse:n .....
.... 212, 832, 848, 18576, 18661,
  18728, 19401, 19566, 19999, 20471
\l_regex_last_char_int .....
..... 20468, 20494, 20598
\l_regex_left_state_int .....
  20116, 20139, 20169, 20176, 20189,
  20199, 20206, 20209, 20210, 20212,
  20213, 20239, 20247, 20250, 20277,
  20328, 20330, 20340, 20360, 20380,
  20382, 20410, 20413, 20416, 20419,
  20431, 20444, 20453, 20477, 20484
\l_regex_left_state_seq .....
..... 20116, 20168, 20175, 20327
\__regex_match:n .....
.... 18693, 20513, 21202, 21212,
  21222, 21231, 21256, 21358, 21388
\l_regex_match_count_int .....
  896, 897, 21179, 21209, 21210, 21215
\__regex_match_init: ..... 20513
\__regex_match_loop: .....
..... 877, 880, 20571, 20594
\__regex_match_once: .....
..... 877, 878, 20530, 20552, 20590
\__regex_match_one_active:n .. 20594
\l_regex_match_success_bool ...
  876, 20510, 20564, 20580, 20587, 20736
\l_regex_max_active_int .....
..... 20502, 20566, 20603,
  20606, 20611, 20714, 20715, 20719
\l_regex_max_pos_int .....
..... 884, 19426, 19427, 19434,
  20088, 20161, 20489, 20527, 20612,
  21257, 21262, 21268, 21377, 21406
\l_regex_max_state_int .....
.. 862, 909, 20113, 20136, 20153,
  20191, 20192, 20198, 20200, 20201,
  20260, 20275, 20339, 20359, 20361,
  20369, 20413, 20419, 20427, 20437,
  20522, 20539, 20544, 20548, 21741
\l_regex_min_active_int . 20502,
  20544, 20566, 20603, 20605, 20611
\l_regex_min_pos_int 884, 19424,
  19433, 20086, 20489, 20524, 20546
\l_regex_min_state_int .. 20113,
  20136, 20153, 20539, 20567, 21740
\l_regex_min_submatch_int .....
..... 896, 899, 902, 20547,
  20549, 21182, 21280, 21393, 21401
\l_regex_mode_int ..... 18975,
  19027, 19035, 19038, 19047, 19050,
  19059, 19067, 19070, 19080, 19081,
  19083, 19085, 19139, 19153, 19155,
  19457, 19461, 19462, 19463, 19490,
  19501, 19617, 19706, 19707, 19735,
  19736, 19792, 19793, 19902, 19948
\__regex_mode_quit_c: .....
..... 19078, 19176, 19590
\__regex_msg_repeated:nnN .....
..... 20043, 20064, 20074, 21704
\__regex_multi_match:n .....
  875, 20574, 21210, 21229, 21237, 21385
\c_regex_no_match_regex .....
..... 18559, 18998, 21109
\c_regex_outer_mode_int .....
  18975, 19038, 19050, 19059, 19067,

```

- 19081, 19139, 19155, 19902, 19948
- _regex_pop_lr_states:
 - 20158, 20166, 20282
- _regex_posix_alnum: [18731](#)
- _regex_posix_alpha: [18731](#)
- _regex_posix_ascii: [18731](#)
- _regex_posix_blank: [18731](#)
- _regex_posix_cntrl: [18731](#)
- _regex_posix_digit: [18731](#)
- _regex_posix_graph: [18731](#)
- _regex_posix_lower: [18731](#)
- _regex_posix_print: [18731](#)
- _regex_posix_punct: [18731](#)
- _regex_posix_space: [18731](#)
- _regex_posix_upper: [18731](#)
- _regex_posix_word: [18731](#)
- _regex_posix_xdigit: [18731](#)
- _regex_prop_: [844](#), [19382](#)
- _regex_prop_d: ... [844](#), [18702](#), [18749](#)
- _regex_prop_h: [18702](#), [18741](#)
- _regex_prop_N: [18702](#), [19410](#)
- _regex_prop_s: [18702](#)
- _regex_prop_v: [18702](#)
- _regex_prop_w:
 - .. [18702](#), [18770](#), [20469](#), [20471](#), [20472](#)
- _regex_push_lr_states:
 - 20156, 20166, 20280
- _regex_query_get:
 - 20570, 20600, 20638
- _regex_query_range:nn [884](#), [20753](#),
 - [20758](#), [20777](#), [20865](#), [21372](#), [21405](#)
- _regex_query_range_loop:ww . [20758](#)
- _regex_query_set:nnn
 - [876](#), [20523](#), [20526](#), [20528](#), [20623](#)
- _regex_query_submatch:n
 - [20775](#), [20913](#), [21312](#)
- _regex_replace_all:nnN [21176](#), [21382](#)
- _regex_replace_once:nnN
 - [21174](#), [21352](#)
- _regex_replacement:n
 - [20816](#), [21357](#), [21387](#)
- _regex_replacement_aux:n ... [20816](#)
- _regex_replacement_balance-
 - one_match:n [212](#),
 - [883](#), [883](#), [20749](#), [20850](#), [21365](#), [21396](#)
- _regex_replacement_c:w [20958](#)
- _regex_replacement_c_A:w [886](#), [21044](#)
- _regex_replacement_c_B:w ... [21047](#)
- _regex_replacement_c_C:w ... [21056](#)
- _regex_replacement_c_D:w ... [21061](#)
- _regex_replacement_c_E:w ... [21064](#)
- _regex_replacement_c_L:w ... [21073](#)
- _regex_replacement_c_M:w ... [21076](#)
- _regex_replacement_c_O:w ... [21079](#)
- _regex_replacement_c_P:w ... [21082](#)
- _regex_replacement_c_S:w ... [21088](#)
- _regex_replacement_c_T:w ... [21096](#)
- _regex_replacement_c_U:w ... [21099](#)
- _regex_replacement_cat:NNN ...
 - [20966](#), [21000](#)
- \l_regex_replacement_category_-
 - seq [20746](#),
 - [20844](#), [20847](#), [20848](#), [20884](#), [21018](#)
- \l_regex_replacement_category_-
 - tl [886](#), [20746](#),
 - [20879](#), [20885](#), [20891](#), [21019](#), [21020](#)
- _regex_replacement_char:nnN ...
 - [893](#), [21039](#),
 - [21046](#), [21053](#), [21063](#), [21070](#), [21075](#),
 - [21078](#), [21081](#), [21085](#), [21098](#), [21101](#)
- \l_regex_replacement_csnames_-
 - int [882](#), [20745](#), [20838](#),
 - [20840](#), [20842](#), [20914](#), [20974](#), [20981](#),
 - [20993](#), [20995](#), [21005](#), [21050](#), [21067](#)
- _regex_replacement_cu_aux:Nw ..
 - [20963](#), [20972](#), [20988](#)
- _regex_replacement_do_one_-
 - match:n . [20751](#), [20863](#), [21370](#), [21404](#)
- _regex_replacement_error:NNN ..
 - [20929](#), [20941](#),
 - [20952](#), [20967](#), [20970](#), [20989](#), [21103](#)
- _regex_replacement_escaped:N ..
 - [20834](#), [20897](#), [21023](#)
- _regex_replacement_exp_not:N ..
 - [889](#), [20757](#), [20963](#)
- _regex_replacement_g:w [20923](#)
- _regex_replacement_g_digits:NN
 - [20923](#)
- _regex_replacement_normal:n ...
 - [20830](#), [20835](#), [20877](#), [20904](#), [20927](#),
 - [20933](#), [20960](#), [20987](#), [20997](#), [21015](#)
- _regex_replacement_put_-
 - submatch:n ... [20902](#), [20909](#), [20948](#)
- _regex_replacement_rbrace:N ...
 - [20828](#), [20947](#), [20991](#)
- _regex_replacement_u:w [20983](#)
- _regex_return:
 - [895](#), [21150](#), [21155](#), [21166](#), [21168](#), [21188](#)
- \l_regex_right_state_int
 - [20116](#), [20142](#), [20159](#), [20171](#),
 - [20178](#), [20190](#), [20199](#), [20200](#), [20239](#),
 - [20246](#), [20252](#), [20265](#), [20275](#), [20277](#),
 - [20330](#), [20334](#), [20345](#), [20359](#), [20368](#),
 - [20380](#), [20384](#), [20388](#), [20391](#), [20396](#),
 - [20399](#), [20402](#), [20410](#), [20424](#), [20427](#),
 - [20430](#), [20433](#), [20437](#), [20453](#), [20484](#)
 - \l_regex_right_state_seq
 - [20116](#), [20170](#), [20177](#), [20332](#)

\l_regex_saved_success_bool . . .
 [876](#), [18691](#), [18698](#), [20510](#)
 _regex_show:N . [19952](#), [21132](#), [21141](#)
 _regex_show_anchor_to_str:N . . .
 [19980](#), [20081](#)
 _regex_show_class:NnnnN
 [19971](#), [20045](#)
 _regex_show_group_aux:nnnnN . . .
 [19966](#), [19968](#), [19970](#), [20036](#)
 _regex_show_item_catcode:NnTF . . .
 [19996](#), [19998](#), [20092](#)
 _regex_show_item_exact_cs:n . . .
 [20003](#), [20105](#)
 \l_regex_show_lines_int
 . . . [19000](#), [20017](#), [20049](#), [20052](#), [20059](#)
 _regex_show_one:n [19960](#),
 [19973](#), [19976](#), [19982](#), [19985](#), [19989](#),
 [19992](#), [20002](#), [20006](#), [20015](#), [20031](#),
 [20038](#), [20042](#), [20055](#), [20071](#), [20110](#)
 _regex_show_pop: [20025](#), [20041](#)
 \l_regex_show_prefix_seq
 [18999](#), [19958](#),
 [19961](#), [20007](#), [20021](#), [20026](#), [20028](#)
 _regex_show_push:n
 [20008](#), [20025](#), [20039](#), [20050](#)
 _regex_show_scope:nn
 [20000](#), [20005](#), [20025](#), [20097](#)
 _regex_single_match:
 [875](#), [18688](#), [20574](#), [21200](#), [21220](#), [21355](#)
 _regex_split:nnN [21178](#), [21234](#)
 _regex_standard_escapechar: . . .
 . . . [18510](#), [18513](#), [18793](#), [19138](#), [20134](#)
 \l_regex_start_pos_int
 [19425](#), [20087](#),
 [20489](#), [20558](#), [20563](#), [20569](#), [21240](#),
 [21252](#), [21265](#), [21268](#), [21342](#), [21406](#)
 \g_regex_state_active_intarray . . .
 [818](#), [874](#), [875](#), [876](#), [20504](#),
 [20542](#), [20655](#), [20658](#), [20666](#), [20693](#)
 \l_regex_step_int
 [817](#), [20501](#), [20545](#), [20596](#),
 [20656](#), [20660](#), [20668](#), [20682](#), [20684](#)
 _regex_store_state:n
 [20567](#), [20707](#), [20710](#)
 _regex_store_submatches:
 [20710](#), [20724](#)
 _regex_submatch_balance:n
 . . . [20750](#), [20781](#), [20854](#), [20917](#), [21304](#)
 \g_regex_submatch_begin_-
 intarray [818](#), [883](#), [20755](#),
 [20778](#), [20803](#), [20811](#), [20872](#), [21185](#),
 [21247](#), [21250](#), [21263](#), [21324](#), [21348](#)
 \g_regex_submatch_end_intarray . . .
 . . . [818](#), [20779](#), [20788](#), [20796](#), [21185](#),
 [21244](#), [21260](#), [21326](#), [21351](#), [21374](#)
 \l_regex_submatch_int
 [818](#), [896](#), [898](#), [899](#), [902](#),
 [20549](#), [21182](#), [21259](#), [21261](#), [21264](#),
 [21266](#), [21269](#), [21281](#), [21321](#), [21325](#),
 [21327](#), [21329](#), [21330](#), [21395](#), [21403](#)
 \g_regex_submatch_prev_intarray . . .
 [818](#), [896](#), [900](#), [20754](#), [20868](#),
 [21185](#), [21242](#), [21258](#), [21328](#), [21341](#)
 \g_regex_success_bool [876](#),
 [18692](#), [18694](#), [18697](#), [20510](#), [20537](#),
 [20579](#), [20588](#), [21190](#), [21320](#), [21359](#)
 \l_regex_success_pos_int
 . . . [20489](#), [20546](#), [20563](#), [20739](#), [21240](#)
 \l_regex_success_submatches_-
 prop . . . [874](#), [900](#), [20499](#), [20740](#), [21332](#)
 _regex_tests_action_cost:n
 [20217](#), [20238](#), [20247](#), [20265](#)
 \g_regex_thread_state_intarray . . .
 [818](#),
 [873](#), [875](#), [875](#), [881](#), [20504](#), [20620](#), [20713](#)
 _regex_tmp:w
 [18532](#), [18534](#), [18538](#), [18540](#), [18549](#),
 [19394](#), [19404](#), [19405](#), [19406](#), [19407](#),
 [19408](#), [19419](#), [19424](#), [19425](#), [19426](#),
 [19427](#), [19428](#), [19433](#), [19434](#), [21161](#),
 [21170](#), [21172](#), [21174](#), [21176](#), [21178](#)
 _regex_toks_clear:N . [18516](#), [20198](#)
 _regex_toks_memcpy:NNn [18521](#), [20370](#)
 _regex_toks_put_left:Nn
 [18530](#), [20181](#), [20352](#), [20353](#)
 _regex_toks_put_right:Nn
 [819](#), [18530](#), [20139](#), [20142](#),
 [20159](#), [20183](#), [20206](#), [20444](#), [20477](#)
 _regex_toks_set:Nn
 [18516](#), [20631](#), [20719](#)
 _regex_toks_use:w
 . . . [18515](#), [20621](#), [20657](#), [20771](#), [21744](#)
 _regex_trace:nnn [20186](#),
 [20516](#), [20533](#), [20649](#), [21720](#), [21743](#)
 _regex_trace_pop:nnN
 [18782](#), [20130](#), [20149](#),
 [20270](#), [20323](#), [20518](#), [20818](#), [21720](#)
 _regex_trace_push:nnN
 [18779](#), [20127](#), [20146](#),
 [20269](#), [20322](#), [20515](#), [20817](#), [21720](#)
 \g_regex_trace_regex_int [21734](#)
 _regex_trace_states:n
 [20129](#), [20148](#), [21735](#)
 _regex_use_state:
 [20647](#), [20670](#), [20696](#)
 _regex_use_state_and_submatches:nn
 [878](#), [20619](#), [20662](#)

- \l_regex_zeroth_submatch_int . . .
 896, 900, 21182,
 21243, 21245, 21248, 21251, 21321,
 21339, 21342, 21366, 21371, 21375
 - \regular_expression 211
 - \relax 14, 21, 39, 43, 49,
 90, 92, 98, 123, 146, 167, 181, 211,
 212, 213, 214, 215, 216, 217, 218,
 219, 220, 221, 224, 225, 226, 227,
 228, 229, 230, 231, 232, 233, 234, 519
 - \relpenalty 520
 - \RequirePackage 149
 - reverse commands:
 - \reverse_if:N . . . 20, 387, 416, 417,
 630, 1402, 4524, 5531, 5533, 5535,
 5537, 10153, 10158, 10162, 10164,
 12478, 16061, 16729, 16752, 18589,
 18590, 18609, 18610, 18617, 18618
 - \right 521
 - right commands:
 - \c_right_brace_str
 . . . 59, 4576, 18901, 19254, 19275,
 19289, 19800, 19804, 19887, 20827
 - \rightghost 988
 - \righthyphenmin 522
 - \rightmarginkern 793
 - \rightskip 523
 - \romannumeral 524
 - round 192
 - \rpcode 794
 - \rule 23049, 23104
- S**
- s@ internal commands:
 - \s@_ 803
 - \saveboxresource 1017
 - \savecatcodetable 960
 - \saveimageresource 1018
 - \savepos 1016
 - \savingshyphcodes 656
 - \savingsdiscards 657
 - scan commands:
 - \scan_align_safe_stop: 27210
 - \scan_new:N
 63, 4697, 4710, 4715, 7874,
 11588, 11591, 11592, 11593, 11594,
 11595, 11596, 11597, 11689, 18065
 - \scan_stop:
 . . . 8, 63, 63, 76, 124, 125, 125, 125,
 251, 265, 301, 313, 314, 316, 328,
 328, 346, 349, 362, 387, 392, 417,
 422, 476, 476, 483, 485, 486, 496,
 518, 547, 547, 548, 554, 626, 630,
 630, 631, 632, 635, 832, 854, 1011,
 1430, 1666, 1684, 1697, 1762, 1767,
 1783, 1788, 1927, 2052, 2070, 2080,
 2098, 2124, 2501, 2767, 2807, 2833,
 2856, 2872, 2998, 3004, 3251, 3275,
 4525, 4707, 5014, 5747, 7459, 7531,
 7735, 7849, 7858, 7867, 9149, 9151,
 9303, 9305, 10009, 10039, 10042,
 10047, 10051, 10055, 10060, 10066,
 10071, 10327, 10399, 10428, 10431,
 10440, 10443, 10448, 10451, 10481,
 10494, 10504, 10533, 10570, 10573,
 10584, 10587, 10592, 10595, 10606,
 10659, 10683, 10722, 10727, 10729,
 10750, 10752, 11574, 11753, 12476,
 12480, 12683, 12700, 13000, 13047,
 13048, 13307, 13350, 13380, 14173,
 17570, 17571, 17687, 18187, 18227,
 18231, 18237, 18239, 18286, 18288,
 18675, 18676, 19011, 19829, 20107,
 20642, 20645, 20672, 21041, 21093,
 21761, 21894, 23045, 23100, 24248,
 24249, 24376, 24379, 24402, 25665,
 25668, 25670, 25847, 26180, 26186
 - scan internal commands:
 - \g__scan_marks_tl . . . 4696, 4700, 4706
 - \scantextokens 961
 - \scantokens 658
 - \scriptfont 525
 - \scriptscriptfont 526
 - \scriptscriptstyle 527
 - \scriptspace 528
 - \scriptstyle 529
 - \scrollmode 530
 - sec 192
 - secd 193
 - seq commands:
 - \c_empty_seq 74, 394, 4724,
 4728, 4732, 4735, 4917, 4948, 4956
 - \l_foo_seq 207
 - \seq_clear:N . . . 65, 65, 74, 4731, 4738,
 4861, 8563, 8626, 9887, 20007, 20848
 - \seq_clear_new:N 65, 4737
 - \seq_concat:NNN . . . 66, 74, 4814, 9895
 - \seq_const_from_clist:Nn . . 241, 24187
 - \seq_count:N 67, 71, 73, 199,
 5075, 5147, 5175, 20847, 24184, 24202
 - \seq_elt:w 393
 - \seq_elt_end: 393
 - \seq_gclear:N
 . . . 65, 790, 4731, 4741, 17809, 24217
 - \seq_gclear_new:N 65, 4737
 - \seq_gconcat:NNN 66, 4814, 9907
 - \seq_get:NN 72, 5228, 20327, 20332
 - \seq_get:NNTF 72, 5234

- \seq_get_left:NN
..... [66](#), [4964](#), [5228](#), [5229](#), [5234](#), [5235](#)
- \seq_get_left:NNTF [68](#), [5034](#)
- \seq_get_right:NN [67](#), [4989](#)
- \seq_get_right:NNTF [68](#), [5034](#)
- \seq_gpop:NN [72](#), [5228](#), [9832](#)
- \seq_gpop:NNTF .. [73](#), [5234](#), [9133](#), [9287](#)
- \seq_gpop_left:NN
..... [67](#), [4975](#), [5232](#), [5233](#), [5238](#), [5239](#)
- \seq_gpop_left:NNTF [68](#), [5042](#)
- \seq_gpop_right:NN [67](#), [5007](#)
- \seq_gpop_right:NNTF [68](#), [5042](#)
- \seq_gpush:Nn
..... [22](#), [73](#), [5208](#), [9160](#), [9315](#), [9815](#)
- \seq_gput_left:Nn
.. [66](#), [4824](#), [5218](#), [5219](#), [5220](#), [5221](#),
[5222](#), [5223](#), [5224](#), [5225](#), [5226](#), [5227](#)
- \seq_gput_right:Nn
[66](#), [4845](#), [9677](#), [9684](#), [9697](#), [9801](#), [9806](#)
- \seq_gremove_all:Nn .. [69](#), [4871](#), [9255](#)
- \seq_gremove_duplicates:N .. [69](#), [4855](#)
- \seq_greverse:N [69](#), [4897](#)
- \seq_gset_eq:NN
.. [65](#), [4735](#), [4743](#), [4858](#), [17783](#), [24199](#)
- \seq_gset_filter:NNn [241](#), [24147](#)
- \seq_gset_from_clist:NN [65](#), [4751](#)
- \seq_gset_from_clist:Nn [65](#), [4751](#)
- \seq_gset_from_function:NnN
..... [242](#), [24177](#)
- \seq_gset_from_inline_x:Nnn
..... [242](#), [17801](#), [24167](#), [24180](#), [24212](#)
- \seq_gset_map:NNn [241](#), [24157](#)
- \seq_gset_split:Nnn
..... [66](#), [4777](#), [9087](#), [9249](#)
- \seq_gshuffle:N [242](#), [24193](#)
- \seq_gsort:Nn [69](#), [4915](#), [17779](#)
- \seq_if_empty:NTF
..... [69](#), [4915](#), [6650](#), [20844](#), [24183](#)
- \seq_if_empty_p:N [69](#), [4915](#)
- \seq_if_exist:NTF
..... [66](#), [4738](#), [4741](#), [4820](#), [5173](#)
- \seq_if_exist_p:N [66](#), [4820](#)
- \seq_if_in:Nn [459](#)
- \seq_if_in:NnTF [69](#),
[73](#), [74](#), [4864](#), [4925](#), [9159](#), [9314](#), [9964](#)
- \seq_item:Nn [67](#),
[210](#), [976](#), [5062](#), [8644](#), [8645](#), [8650](#), [24184](#)
- \seq_log:N [75](#), [5240](#)
- \seq_map_break: [70](#),
[241](#), [241](#), [5086](#), [5097](#), [5132](#), [5143](#), [11379](#)
- \seq_map_break:n [71](#), [404](#),
[5086](#), [8583](#), [8597](#), [9744](#), [17780](#), [17783](#)
- \seq_map_function:NN
..... [4](#), [70](#), [239](#), [5090](#),
[5250](#), [6656](#), [8648](#), [9898](#), [20021](#), [20100](#)
- \seq_map_inline:Nn
..... [70](#), [70](#), [74](#), [975](#), [4862](#), [5128](#),
[8578](#), [9713](#), [9743](#), [11372](#), [17780](#), [17783](#)
- \seq_map_variable:NNn [70](#), [5135](#)
- \seq_mapthread_function:NNN
..... [241](#), [24125](#)
- \seq_new:N [4](#), [65](#),
[65](#), [4725](#), [4738](#), [4741](#), [4854](#), [5254](#),
[5255](#), [5256](#), [5257](#), [6801](#), [7247](#), [7250](#),
[8533](#), [8534](#), [9085](#), [9246](#), [9668](#), [9692](#),
[9706](#), [9708](#), [10793](#), [17641](#), [18556](#),
[18999](#), [20118](#), [20119](#), [20747](#), [24197](#)
- \seq_pop:NN
..... [72](#), [5228](#), [20175](#), [20177](#), [20884](#)
- \seq_pop:NNTF [73](#), [5234](#)
- \seq_pop_left:NN
..... [67](#), [4975](#), [5230](#), [5231](#), [5236](#), [5237](#)
- \seq_pop_left:NNTF [68](#), [5042](#)
- \seq_pop_right:NN
..... [67](#), [5007](#), [19958](#), [20028](#)
- \seq_pop_right:NNTF [68](#), [5042](#)
- \seq_push:Nn
.. [73](#), [5208](#), [5215](#), [20168](#), [20170](#), [21018](#)
- \seq_put_left:Nn [66](#),
[4824](#), [5208](#), [5209](#), [5210](#), [5211](#), [5212](#),
[5213](#), [5214](#), [5215](#), [5216](#), [5217](#), [8573](#)
- \seq_put_right:Nn [66](#), [73](#), [74](#), [4845](#),
[4865](#), [8634](#), [9960](#), [9965](#), [19961](#), [20026](#)
- \seq_rand_item:N [241](#), [24181](#)
- \seq_remove_all:Nn
[66](#), [69](#), [73](#), [74](#), [4871](#), [6827](#), [9968](#), [9972](#)
- \seq_remove_duplicates:N
..... [69](#), [73](#), [74](#), [4855](#), [9896](#)
- \seq_reverse:N [69](#), [399](#), [4897](#)
- \seq_set_eq:NN
[65](#), [74](#), [4732](#), [4743](#), [4856](#), [17780](#), [24198](#)
- \seq_set_filter:NNn
..... [241](#), [975](#), [20095](#), [24147](#)
- \seq_set_from_clist:NN [65](#), [4751](#), [6826](#)
- \seq_set_from_clist:Nn
[65](#), [104](#), [976](#), [4751](#), [9891](#), [9905](#), [11301](#)
- \seq_set_from_function:NnN
..... [242](#), [21278](#), [24177](#)
- \seq_set_from_inline_x:Nnn
..... [242](#), [976](#), [24167](#), [24178](#)
- \seq_set_map:NNn
..... [241](#), [20108](#), [21296](#), [24157](#)
- \seq_set_split:Nnn
.. [66](#), [4777](#), [7248](#), [7251](#), [20094](#), [20107](#)
- \seq_show:N [75](#), [505](#), [5240](#)
- \seq_shuffle:N [242](#), [24193](#)

- \seq_sort:Nn 69, 200, 4915, 17779
- \seq_use:Nn 72, 5171, 20111
- \seq_use:Nnnn 71, 5171
- \g_tmpa_seq 75, 5254
- \l_tmpa_seq 75, 5254
- \g_tmpb_seq 75, 5254
- \l_tmpb_seq 75, 5254
- seq internal commands:
- __seq_count:w 406, 5147
- __seq_count_end:w 406, 5147
- __seq_get_left:wnw 4964
- __seq_get_right_end:NnN 4989
- __seq_get_right_loop:nw . . 402, 4989
- __seq_if_in: 4925
- \l_seq_internal_a_int 24193
- \l_seq_internal_a_tl
- 395, 4721, 4785, 4789, 4795,
- 4800, 4802, 4886, 4891, 4929, 4933
- \l_seq_internal_b_int 24193
- \l_seq_internal_b_tl
- 4721, 4882, 4886, 4932, 4933
- \g_seq_internal_seq 24193
- __seq_item:n
- 393, 393, 393, 393, 397, 400, 401,
- 402, 404, 405, 405, 405, 406, 407,
- 974, 975, 975, 4716, 4828, 4836,
- 4846, 4848, 4853, 4903, 4904, 4906,
- 4911, 4930, 4969, 4972, 4982, 4997,
- 5000, 5013, 5014, 5025, 5069, 5078,
- 5096, 5099, 5109, 5114, 5120, 5124,
- 5154, 5155, 5156, 5157, 5158, 5159,
- 5160, 5161, 5166, 5167, 5182, 5197,
- 5200, 5203, 24163, 24173, 24174, 24209
- __seq_item:nN 5062
- __seq_item:nwn 5062
- __seq_item:wNn 5062
- __seq_map_function:NNn 5090
- __seq_map_function:Nw
- 5093, 5099, 5103
- __seq_mapthread_function:Nnnwnn 24125
- __seq_mapthread_function:wNN . 24125
- __seq_mapthread_function:wNw . 24125
- __seq_pop:NNNN
- 4946, 4976, 4978, 5008, 5010
- __seq_pop_item_def:
- 393, 393, 4893,
- 5106, 5132, 5143, 24155, 24165, 24175
- __seq_pop_left:NNN . 4975, 5044, 5047
- __seq_pop_left:wnwNNN 4975
- __seq_pop_right:NNN
- 398, 5007, 5050, 5053
- __seq_pop_right_loop:nn 5007
- __seq_pop_TF:NNNN 403, 4946,
- 5035, 5037, 5044, 5047, 5050, 5053
- __seq_push_item_def: 5106
- __seq_push_item_def:n
- 393, 393, 4877,
- 5106, 5130, 5137, 24153, 24163, 24173
- __seq_put_left_aux:w 397, 4824
- __seq_remove_all_aux:NNn 4871
- __seq_remove_duplicates:NN . . 4855
- \l_seq_remove_seq
- 4854, 4861, 4864, 4865, 4867
- __seq_reverse:NN 4897
- __seq_reverse_item:nw 399
- __seq_reverse_item:nwn 4897
- __seq_set_filter:NNNn 24147
- __seq_set_from_inline_x:NNnn . 24167
- __seq_set_map:NNNn 24157
- __seq_set_split:NNnn 4777
- __seq_set_split_auxi:w . . . 395, 4777
- __seq_set_split_auxii:w . . 395, 4777
- __seq_set_split_end: 395, 4777
- __seq_show:NN 5240
- __seq_shuffle:NN 24193
- __seq_shuffle_item:n 24193
- __seq_tmp:w
- 4723, 4903, 4906, 5013, 5025
- __seq_use:NNnNnn 5171
- __seq_use:nwwn 5171
- __seq_use:nwwwwnwn 5171
- __seq_use_setup:w 5171
- __seq_wrap_item:n
- 395, 975, 4754, 4759, 4764, 4769,
- 4786, 4811, 4853, 4889, 24153, 24190
- \setbox 531
- \setfontid 962
- \setlanguage 532
- \setrandomseed 1019
- \sfcode 183, 533
- \sffamily 23038
- \shapemode 963
- \shellescape 876
- \Shipout 1264
- \shipout 534, 1251, 1252
- \ShortText 75, 116, 133
- \show 535
- \showbox 536
- \showboxbreadth 537
- \showboxdepth 538
- \showgroups 659
- \showifs 660
- \showlists 539
- \showmode 1218
- \showthe 540
- \ShowTokens 201

- \showtokens 661
- sign 192
- sin 192
- sind 193
- \sjis 1219
- \skewchar 541
- \skip 542, 7609
- skip commands:
 - \c_max_skip 161, 10515
 - \skip_add:Nn 159, 10438
 - \skip_const:Nn 158, 561, 10395, 10515, 10516
 - \skip_eval:n 160, 160, 160, 160, 10399, 10456, 10478, 10510, 10514
 - \skip_gadd:Nn 159, 10438
 - \skip_gset:N 169, 11193
 - \skip_gset:Nn 159, 557, 10426
 - \skip_gset_eq:NN 159, 10434
 - \skip_gsub:Nn 159, 10438
 - \skip_gzero:N 159, 10402, 10411
 - \skip_gzero_new:N 159, 10408
 - \skip_horizontal:N 161, 10485
 - \skip_horizontal:n 161, 10485, 25889, 26210, 26348, 26827
 - \skip_if_eq:nnTF 160, 10454
 - \skip_if_eq_p:nn 160, 10454
 - \skip_if_exist:NTF 159, 10409, 10411, 10414
 - \skip_if_exist_p:N 159, 10414
 - \skip_if_finite:nTF 160, 10460, 24246
 - \skip_if_finite_p:n 160, 10460
 - \skip_log:N 161, 10511
 - \skip_log:n 161, 10511
 - \skip_new:N 158, 159, 10387, 10398, 10409, 10411, 10517, 10518, 10519, 10520
 - \skip_set:N 169, 11193
 - \skip_set:Nn 159, 10426
 - \skip_set_eq:NN 159, 10434
 - \skip_show:N 160, 10507
 - \skip_show:n 160, 560, 10509
 - \skip_split_finite_else_action:nnNN 242, 24244
 - \skip_sub:Nn 159, 10438
 - \skip_use:N 160, 160, 10472, 10481, 10482
 - \skip_vertical:N 161, 10485
 - \skip_vertical:n 161, 10485
 - \skip_zero:N 159, 159, 162, 10402, 10409
 - \skip_zero_new:N 159, 10408
 - \g_tmpa_skip 161, 10517
 - \l_tmpa_skip 161, 10517
 - \g_tmpb_skip 161, 10517
 - \l_tmpb_skip 161, 10517
 - \c_zero_skip 161, 546, 10013, 10016, 10403, 10405, 10515, 24252, 24253
 - skip internal commands:
 - _skip_if_finite:wwNw 10460
 - _skip_tmp:w 558, 10418, 10426, 10429, 10438, 10441, 10446, 10449, 10460, 10477, 10554, 10568, 10571, 10582, 10585, 10590, 10593
 - \skipdef 543
 - sort commands:
 - \sort_ordered: 18058
 - \sort_return_same: 200, 200, 793, 17861, 18058, 18059
 - \sort_return_swapped: 200, 200, 793, 17861, 18060, 18061
 - \sort_reversed: 18058
 - sort internal commands:
 - _sort:nnNnn 794, 795
 - \l__sort_A_int 792, 17651, 17656, 17663, 17666, 17675, 17826, 17831, 17834, 17854, 17877, 17896, 17898, 17899
 - \l__sort_B_int 792, 792, 17651, 17831, 17835, 17843, 17845, 17846, 17886, 17887, 17896, 17897, 17906, 17907, 17909
 - \l__sort_begin_int 787, 792, 17649, 17823, 17899, 17909
 - \l__sort_block_int 787, 787, 791, 17648, 17658, 17663, 17667, 17670, 17675, 17676, 17753, 17814, 17817, 17824, 17827
 - \l__sort_C_int 792, 792, 17651, 17832, 17836, 17843, 17844, 17855, 17878, 17886, 17888, 17889, 17906, 17908
 - _sort_compare:nn 789, 793, 17752, 17853
 - _sort_compute_range: 786, 787, 788, 17680, 17740
 - _sort_copy_block: 792, 17833, 17841
 - _sort_disable_toksdef: 17739, 18006
 - _sort_disabled_toksdef:n 18006
 - \l__sort_end_int 787, 791, 792, 792, 17649, 17815, 17823, 17824, 17825, 17826, 17827, 17828, 17829, 17846
 - _sort_error: 18000, 18013, 18032
 - _sort_i:nnnnNn 796
 - \g__sort_internal_seq 789, 790, 17641, 17801, 17808, 17809
 - \g__sort_internal_tl 17641, 17764, 17767, 17768
 - \l__sort_length_int 786, 787, 17643, 17750, 17814

- _sort_level: 789, 799, 17754, 17812, 18004
- _sort_loop:wNn 795, 796
- _sort_main:NNNn 790, 790, 17737, 17763, 17800
- \l_sort_max_int 786, 787, 17643, 17660, 17734, 17744
- \c_sort_max_length_int 17680
- _sort_merge_blocks: 17816, 17821, 18003
- _sort_merge_blocks_aux: 792, 17837, 17851, 17892, 17902, 18002
- _sort_merge_blocks_end: 794, 17900, 17904
- \l_sort_min_int 786, 787, 789, 790, 17643, 17657, 17665, 17683, 17699, 17707, 17720, 17732, 17741, 17751, 17765, 17804, 17815, 18030, 18031
- _sort_quick_cleanup:w 17914
- _sort_quick_end:nnTFNn 797, 798, 17934, 17974
- _sort_quick_only_i:NnnnnNn . 17939
- _sort_quick_only_i_end:nnwnw . 17950, 17974
- _sort_quick_only_ii:NnnnnNn . 17939
- _sort_quick_only_ii_end:nnwnw . 17957, 17974
- _sort_quick_prepare:Nnnn . 17914
- _sort_quick_prepare_end:NNNnw . 17914
- _sort_quick_single_end:nnwnw . 17943, 17974
- _sort_quick_split:NnNn 796, 797, 17934, 17939, 17979, 17986, 17992, 17994
- _sort_quick_split_end:nnwnw . 17964, 17971, 17974
- _sort_quick_split_i:NnnnnNn . 795, 17939
- _sort_quick_split_ii:NnnnnNn 17939
- _sort_redefine_compute_range: . 17680
- _sort_return_mark:N 17857, 17858, 17861
- _sort_return_none_error: 793, 17859, 17861
- _sort_return_same: 17865, 17879, 17884
- _sort_return_swapped: 17871, 17894
- _sort_return_two_error:w . 17861
- _sort_seq:NNNNn 789, 17779
- _sort_shrink_range: 787, 788, 17654, 17685, 17701, 17709, 17722
- _sort_shrink_range_loop: . 17654
- _sort_tl:NNn 789, 17756
- _sort_tl_toks:w 790, 17756
- _sort_too_long_error:NNw 17745, 18025
- \l_sort_top_int 786, 789, 790, 792, 792, 17643, 17741, 17744, 17747, 17748, 17751, 17773, 17804, 17825, 17828, 17829, 17832, 17889, 18031
- \l_sort_true_max_int 786, 787, 17643, 17657, 17670, 17684, 17700, 17708, 17721, 17733, 18030
- sp 195
- spac commands:
 - \spac_directions_normal_body_dir 1391
 - \spac_directions_normal_page_dir 1392
- \space 55
- \spacefactor 544
- \spaceskip 545
- \span 546
- \special 547
- \splitbotmark 548
- \splitbotmarks 662
- \splitdiscards 663
- \splitfirstmark 549
- \splitfirstmarks 664
- \splitmaxdepth 550
- \splittopskip 551
- sqrt 194
- \SS 25309
- \ss 25309
- str commands:
 - \c_ampersand_str 59, 4576
 - \c_at_sign_str 59, 4576
 - \c_backslash_str 59, 4576, 18811, 19373
 - \c_circumflex_str 59, 4576
 - \c_colon_str 59, 4576, 7522, 7627, 7633, 10887
 - \c_dollar_str 59, 4576
 - \c_hash_str 59, 4576, 23517, 23549, 23553, 26812, 26972, 26979, 27019
 - \c_percent_str 59, 4576, 27105, 27106, 27107
 - \str_case:nn . 52, 4134, 9023, 19522
 - \str_case:nnn 27212, 27214
 - \str_case:nnTF . 52, 551, 4134, 4139, 4144, 8917, 11043, 20084, 24931, 27213, 27215
 - \str_case_x:nn 52, 4134
 - \str_case_x:nnn 27216
 - \str_case_x:nnTF . 52, 4134, 4170, 4175, 19252, 27217

- \str_clear:N 10456, 10930, 18859, 18881, 18890,
- 49, 49, 3965, 9752, 9859, 9871
- \str_clear_new:N 19274, 19288, 19868, 20925, 20985,
- \str_concat:NNN 21576, 21706, 23517, 23549, 23551
- \str_const:Nn \str_if_eq_x_p:nn
- 51, 4098, 6560, 6570, 6572
- \str_count:N 54, 4466, 9374, 9439, 18479
- \str_count:n 54, 4466, 18473
- \str_count_ignore_spaces:n \str_if_exist:Ntf 50, 4068
- 54, 385, 4466, 18094
- \str_count_spaces:N 54, 4446
- \str_count_spaces:n 54, 385, 4446, 4472
- \str_fold_case:n \str_if_exist_p:N 50, 4068
- 57, 58, 244, 250, 4534, 12669
- \str_gclear:N 49, 3965
- \str_gclear_new:N 3965
- \str_gconcat:NNN 49, 3965
- \str_gput_left:Nn 49, 3992
- \str_gput_right:Nn 49, 3992
- \str_gremove_all:Nn 50, 4062
- \str_gremove_once:Nn 50, 4056
- \str_greplace_all:Nnn 50, 4016, 4065
- \str_greplace_once:Nnn 50, 4016, 4059
- \str_gset:Nn \str_if_in:NnTF 51, 4120
- 49, 3992, 9661, 9840, 9841, 9842
- \str_gset_eq:NN \str_if_in:nnTF . 51, 3034, 3048, 4120
- 49, 3965, 9666, 9823, 9824, 9825
- \str_head:N 55, 386, 4504
- \str_head:n 55,
- 351, 368, 386, 3324, 3807, 3852, 4504
- \str_head_ignore_spaces:n .. 55, 4504
- \str_if_empty:Ntf 50,
- 4068, 9118, 9754, 9758, 9781, 9794,
- 9853, 9978, 24004, 24010, 24370, 24397
- \str_if_empty_p:N 50, 4068
- \str_if_eq:NN 377
- \str_if_eq:nn 128, 487
- \str_if_eq:NNTF 51, 4112
- \str_if_eq:nnTF 51, 52, 52, 130, 240,
- 398, 3495, 4098, 4161, 4879, 7949,
- 8382, 8594, 10843, 10861, 11376,
- 12538, 12612, 19227, 19323, 19502,
- 19539, 19559, 19628, 19663, 19680,
- 19742, 21012, 23471, 23488, 23499,
- 23528, 24697, 24716, 24734, 24744,
- 24757, 25774, 25777, 25780, 25783
- \str_if_eq_p:NN 51, 4112
- \str_if_eq_p:nn 51, 4098
- \str_if_eq_x:nn 494
- \str_if_eq_x:nnTF \str_item:Nn 55, 4308
- 51, 477, 2312, 4098,
- 4189, 7525, 7581, 8026, 8111, 8637,
- \str_item:nn 55, 381, 385, 4308
- \str_item_ignore_spaces:nn \str_lower_case:n 57, 244, 4534
- 55, 381, 4308
- \str_map_break: 53, 4195
- \str_map_break:n ... 53, 54, 3038, 4195
- \str_map_function:NN \str_map_function:nn 52, 52, 53, 53, 4195
- 52, 52, 379, 4195
- \str_map_inline:Nn .. 53, 53, 53, 4195
- \str_map_inline:nn ... 53, 3032, 4195
- \str_map_variable:NNn 53, 4195
- \str_map_variable:nnn 53, 4195
- \str_new:N \str_put_left:Nn 49, 3992, 9856
- 48, 49, 3965, 4588, 4589, 4590,
- 4591, 9114, 9282, 9654, 9655, 9656,
- 9701, 9702, 9703, 9704, 9705, 24409
- \str_put_right:Nn 49, 3992, 9763
- \str_range:Nnn 56, 4369
- \str_range:nnn .. 56, 385, 4369, 18476
- \str_range_ignore_spaces:nnn 56, 4369
- \str_remove_all:Nn 50, 50, 4062
- \str_remove_once:Nn 50, 4056
- \str_replace_all:Nnn . 50, 4016, 4063
- \str_replace_once:Nnn 50, 4016, 4057
- \str_set:Nn \str_show:N 58, 4592
- 49, 50, 3992, 4254, 9722, 9726,
- 9735, 9853, 9868, 9875, 23953, 23964
- \str_set_eq:NN .. 49, 3965, 9766, 9858
- \str_show:N 58, 4592
- \str_show:n 58, 4592
- \str_tail:N 55, 4519
- \str_tail:n 55, 806, 4519, 9875
- \str_tail_ignore_spaces:n .. 55, 4519
- \str_upper_case:n 57, 244, 4534
- \str_use:N 54, 3965
- \c_tilde_str 59, 4576
- \g_tmpa_str 59, 4588
- \l_tmpa_str 50, 59, 4588
- \g_tmpb_str 59, 4588
- \l_tmpb_str 59, 4588
- \c_underscore_str 59, 4576

- str internal commands:
- `__str_case:nnTF` [4134](#)
 - `__str_case:nw` [4134](#)
 - `__str_case_end:nw` [4134](#)
 - `__str_case_x:nnTF` [4134](#)
 - `__str_case_x:nw` [4134](#)
 - `__str_change_case:nn` [4534](#)
 - `__str_change_case_aux:nn` [4534](#)
 - `__str_change_case_char:nN` [4534](#)
 - `__str_change_case_end:nw` [4534](#)
 - `__str_change_case_end:wn` [4553](#), [4571](#)
 - `__str_change_case_loop:nw` [4534](#)
 - `__str_change_case_output:nw` . [4534](#)
 - `__str_change_case_result:n` . [4534](#)
 - `__str_change_case_space:n` [4534](#)
 - `__str_collect_delimit_by_q-stop:w` [4397](#), [4420](#)
 - `__str_collect_end:nnnnnnnw` [384](#), [4420](#)
 - `__str_collect_end:wn` [4420](#)
 - `__str_collect_loop:wn` [4420](#)
 - `__str_collect_loop:wnNNNNNNN` . [4420](#)
 - `__str_count:n` . [385](#), [4324](#), [4384](#), [4466](#)
 - `__str_count_aux:n` [4466](#)
 - `__str_count_loop:NNNNNNNN` . [4466](#)
 - `__str_count_spaces_loop:w` [4446](#)
 - `__str_escape_x:n` [4076](#)
 - `__str_head:w` [386](#), [4504](#)
 - `__str_if_eq_x:nn` [4076](#), [4101](#), [4109](#), [4115](#)
 - `__str_item:nn` [381](#), [4308](#)
 - `__str_item:w` [381](#), [4308](#)
 - `__str_map_function:Nn` [379](#), [4195](#)
 - `__str_map_function:w` [378](#), [379](#), [4195](#)
 - `__str_map_inline:NN` [4195](#)
 - `__str_map_variable:NnN` [4195](#)
 - `__str_range:nnn` [4369](#)
 - `__str_range:nnw` [4369](#)
 - `__str_range:w` [4369](#)
 - `__str_range_normalize:nn` [4392](#), [4393](#), [4401](#)
 - `__str_replace:NNNnn` [4016](#)
 - `__str_replace_aux:NNNnnn` [4016](#)
 - `__str_replace_next:w` [4016](#)
 - `__str_skip_end:NNNNNNNN` .. [382](#), [4348](#)
 - `__str_skip_end:w` [4348](#)
 - `__str_skip_exp_end:w` [382](#), [384](#), [4335](#), [4344](#), [4348](#), [4399](#)
 - `__str_skip_loop:wNNNNNNNN` ... [4348](#)
 - `__str_tail_auxi:w` [4519](#)
 - `__str_tail_auxii:w` [387](#), [4519](#)
 - `__str_tmp:n` [3966](#), [3972](#), [3975](#), [3993](#), [4003](#), [4006](#)
 - `__str_to_other_end:w` [380](#), [4263](#)
 - `__str_to_other_fast_end:w` ... [4286](#)
 - `__str_to_other_fast_loop:w` [4288](#), [4297](#), [4304](#)
 - `__str_to_other_loop:w` [380](#), [4263](#)
 - `\strcmp` [40](#)
 - `\string` [552](#)
 - `\suppressfontnotfounderror` [813](#)
 - `\suppressifcsnameerror` [964](#)
 - `\suppresslongerror` [966](#)
 - `\suppressmathparerror` [967](#)
 - `\suppressoutererror` [969](#)
 - `\suppressprimitiveerror` [970](#)
 - `\synctex` [795](#)
- sys commands:
- `\c_sys_day_int` [102](#), [6520](#)
 - `\c_sys_engine_str` [102](#), [6543](#)
 - `\sys_gset_rand_seed:n` [195](#), [243](#), [24263](#)
 - `\c_sys_hour_int` [102](#), [6520](#)
 - `\sys_if_engine luatex:TF` [102](#), [231](#), [6543](#), [7317](#), [7319](#), [11950](#), [23298](#), [23959](#), [24267](#), [24282](#), [24284](#), [24297](#), [27189](#), [27191](#), [27193](#)
 - `\sys_if_engine luatex_p:` ... [102](#), [6543](#), [9241](#), [23458](#), [24435](#), [24646](#), [24675](#), [24857](#), [25042](#), [25084](#), [27187](#)
 - `\sys_if_engine pdftex:TF` [102](#), [6543](#), [27197](#), [27199](#), [27201](#)
 - `\sys_if_engine pdftex_p:` [102](#), [6543](#), [25120](#), [27195](#)
 - `\sys_if_engine ptex:TF` [102](#), [6543](#)
 - `\sys_if_engine ptex_p:` [102](#), [6543](#), [18112](#)
 - `\sys_if_engine uptex:TF` ... [102](#), [6543](#)
 - `\sys_if_engine uptex_p:` [102](#), [6543](#), [18113](#), [25121](#)
 - `\sys_if_engine xetex:TF` .. [4](#), [102](#), [6543](#), [7318](#), [23984](#), [27229](#), [27231](#), [27233](#)
 - `\sys_if_engine xetex_p:` [102](#), [6543](#), [23458](#), [24435](#), [24647](#), [24676](#), [24858](#), [25043](#), [25085](#), [27227](#)
 - `\sys_if_output_dvi:TF` [103](#), [6562](#)
 - `\sys_if_output_dvi_p:` [103](#), [6562](#)
 - `\sys_if_output_pdf:TF` [103](#), [6562](#)
 - `\sys_if_output_pdf_p:` [103](#), [6562](#)
 - `\sys_if_rand_exist:TF` ... [243](#), [24258](#)
 - `\sys_if_rand_exist_p:` ... [243](#), [24258](#)
 - `\sys_if_shell:` [243](#)
 - `\sys_if_shell:TF` [243](#), [24276](#)
 - `\sys_if_shell_p:` [243](#), [24276](#)
 - `\sys_if_shell_restricted:TF` [243](#), [24276](#)
 - `\sys_if_shell_restricted_p:` [243](#), [24276](#)

- `\sys_if_shell_unrestricted:TF` . . .
 [243](#), [24276](#)
- `\sys_if_shell_unrestricted_p:` . . .
 [243](#), [24276](#)
- `\c_sys_jobname_str`
 [102](#), [148](#), [6510](#), [27167](#)
- `\c_sys_minute_int` [102](#), [6520](#)
- `\c_sys_month_int` [102](#), [6520](#)
- `\c_sys_output_str` [103](#), [6562](#)
- `\sys_rand_seed:` . [195](#), [242](#), [243](#), [24260](#)
- `\c_sys_shell_escape_int`
 [243](#), [24265](#), [24277](#), [24279](#), [24281](#)
- `\sys_shell_now:n` [244](#), [24284](#)
- `\sys_shell_shipout:n` [244](#), [24297](#)
- `\c_sys_year_int` [102](#), [6520](#)
- sys internal commands:
 - `_sys_const:nn` [6527](#), [6559](#),
 [6569](#), [6571](#), [24258](#), [24276](#), [24278](#), [24280](#)
 - `\c_sys_shell_stream_int`
 [24282](#), [24294](#), [24307](#)
- syst commands:
 - `\c_syst_last_allocated_toks` . . [17714](#)
- T**
- `\t` [25318](#)
- `\tabskip` [553](#)
- `\tagcode` [796](#)
- `\tan` [192](#)
- `\tand` [193](#)
- `\tate` [1220](#)
- `\tbaselineshift` [1221](#)
- `\temp` . [163](#), [169](#), [174](#), [177](#), [178](#), [185](#), [190](#), [193](#)
- \TeX and \LaTeX 2 ϵ commands:
 - `\@` [4577](#)
 - `\@end` [284](#), [1239](#), [1240](#)
 - `\@hyph` [1243](#)
 - `\@input` [1244](#)
 - `\@italiccorr` [1245](#)
 - `\@shipout` [1247](#), [1248](#)
 - `\@tracingfonts` [285](#)
 - `\@underline` [1246](#)
 - `\@addtofilelist` [9805](#)
 - `\@currname` [537](#), [9665](#), [9666](#)
 - `\@currnamestack` [9688](#), [9689](#)
 - `\@filelist` [149](#), [538](#),
 [541](#), [543](#), [543](#), [9804](#), [9889](#), [9892](#), [9906](#)
 - `\@firstoftwo` [297](#)
 - `\@ifpackageloaded` [25706](#), [25764](#)
 - `\@secondoftwo` [297](#)
 - `\@tempa` [143](#), [145](#), [1255](#), [1269](#), [1272](#)
 - `\@tfor` [284](#), [1255](#)
 - `\@unexpandable@protect` [631](#)
 - `\AtBeginDocument` [284](#)
 - `\botmark` [479](#)
 - `\box` [216](#), [818](#)
 - `\char` [127](#)
 - `\chardef` [121](#), [121](#), [412](#)
 - `\conditionally@traceoff` [530](#)
 - `\copy` [216](#)
 - `\count` [127](#), [788](#), [818](#)
 - `\cr` [445](#)
 - `\CROP@shipout` [1256](#)
 - `\csname` [16](#)
 - `\csstring` [312](#)
 - `\current@color`
 [253](#), [1014](#), [25710](#), [25715](#), [25720](#), [25769](#)
 - `\currentgrouplevel` [324](#), [972](#)
 - `\currentgrouptype` [324](#), [972](#)
 - `\def` [127](#)
 - `\detokenize` [41](#)
 - `\dimen` [477](#)
 - `\dimendef` [477](#)
 - `\directlua` [231](#)
 - `\dp` [217](#), [632](#), [633](#)
 - `\dup@shipout` [1257](#)
 - `\e@alloc@top` [788](#), [17700](#)
 - `\edef` [1](#), [344](#)
 - `\endcsname` [16](#)
 - `\endinput` [136](#)
 - `\endlinechar` . [37](#), [37](#), [143](#), [350](#), [351](#), [479](#)
 - `\endtemplate` [101](#), [445](#)
 - `\errhelp` [499](#), [500](#)
 - `\errmessage` [499](#), [500](#)
 - `\errorcontextlines` . [289](#), [372](#), [500](#), [914](#)
 - `\escapechar` [41](#), [312](#), [324](#), [529](#)
 - `\etex_unexpanded:D` [335](#)
 - `\everyeof` [349](#), [351](#)
 - `\everypar` [235](#), [959](#)
 - `\expandafter` [30](#), [31](#)
 - `\expanded` [235](#), [277](#)
 - `\fi` [126](#)
 - `\firstmark` [336](#), [479](#)
 - `\font` [126](#)
 - `\fontdimen` [199](#), [213](#), [784](#), [785](#), [786](#)
 - `\frozen@everydisplay` [1241](#)
 - `\frozen@everymath` [1242](#)
 - `\futurelet`
 [445](#), [481](#), [483](#), [802](#), [804](#), [806](#), [806](#)
 - `\global` [264](#)
 - `\GPTorg@shipout` [1258](#)
 - `\halign` [101](#), [445](#), [471](#), [959](#)
 - `\hskip` [161](#)
 - `\ht` [217](#), [632](#), [633](#)
 - `\hyphen` [479](#)
 - `\hyphenchar` [784](#)
 - `\ifcase` [89](#)
 - `\ifdim` [164](#)
 - `\ifeof` [148](#)

- \iffalse 94
- \ifhbox 224
- \ifnum 88
- \ifodd 89, 1011
- \iftrue 94
- \ifvbox 224
- \ifvoid 225
- \ifx 20, 261
- \indent 959
- \infty 188
- \input@path 149, 9746, 9748
- \italiccorr 479
- \jobname 102
- \l@expl@check@declarations@bool 1639
- \l@expl@enable@debug@bool 1531
- \l@expl@log@functions@bool ... 1716
- \lastnamedcs 315
- \lccode 261, 432, 805, 809, 956
- \leavevmode 235
- \let 264
- \lcharcode 468
- \LL@shipout 1259
- \loctoks 788
- \long 127
- \lower 960
- \lowercase 891, 892, 892
- \luaescapestring 232
- \m@ne 433, 6150
- \makeatletter 6
- \mathchar 127
- \mathchardef 121, 412
- \meaning 14,
118, 126, 127, 477, 477, 483, 804, 1011
- \mem@oldshipout 1260
- \newif 94
- \newlinechar 37,
37, 289, 315, 350, 351, 372, 500, 528
- \newread 521
- \newtoks 200, 799, 817
- \newwrite 525
- \noexpand 30, 126
- \nullfont 479
- \number 88, 685
- \opem@shipout 1261
- \or 89
- \outer .. 127, 261, 521, 525, 1011, 1050
- \par 950
- \parindent 235
- \pdfmapfile 286
- \pdfmapline 286
- \pdfstrcmp . xii, 258, 259, 261, 275, 955
- \pdfTeX_mdifivesum:D 969
- \pdfuniformdeviate 195
- \pgfpages@originalshipout 1262
- \pgfsys@... 253
- \pi 188
- \pr@shipout 1263
- \primitive 284
- \protect 531, 630, 630, 631, 990
- \protected 127
- \ProvidesClass 6
- \ProvidesFile 6
- \ProvidesPackage 6
- \quitvmode 959
- \read 143
- \readline 143
- \relax 20, 126, 261, 308,
313, 324, 433, 433, 590, 592, 616, 646
- \RequirePackage 6, 261, 537
- \reserveinserts 261
- \robustify 245
- \romannumeral 32, 589
- \scantokens 349, 350, 351
- \sfcode 262
- \shipout 284
- \show 15, 46, 324
- \showbox 913
- \showthe 324, 431, 556, 560, 564
- \showtokens 46, 372, 505
- \sin 188
- \skip 809, 810
- \space 479
- \special 1013
- \splitbotmark 479
- \splitfirstmark 479
- \strcmp 258, 275
- \string 118, 804, 806, 807
- \synctex 275
- \tenrm 126
- \tex_lowercase:D 471
- \the 79, 126, 156, 160, 163, 328
- \toks xx, 88, 200,
242, 786, 787, 787, 788, 789, 790,
790, 792, 792, 793, 794, 794, 799,
803, 804, 805, 807, 809, 810, 812,
817, 818, 819, 819, 819, 819, 820,
824, 862, 869, 869, 874, 874, 874,
875, 876, 879, 881, 884, 891, 909, 976
- \toksdef 799
- \topmark 127, 479
- \tracingfonts 285
- \tracingonline 914
- \typeout 531
- \uccode 956
- \Ucharcat 470, 471
- \ucharcat@table 52, 55
- \unexpanded 30,
42, 42, 43, 46, 67, 71, 72, 107, 110,

- 110, 112, 130, 236, 240, 241, 244,
 248, 248, 249, 344, 366, 367, 417, 471
 \unhbox 220
 \unhcopy 220
 \uniformdeviate 195
 \unless 20
 \unvbox 222
 \unvcopy 222
 \uppercase 891
 \usepackage 537
 \valign 445
 \verso@orig@shipout 1265
 \vskip 161
 \vsplit 221
 \vtop 931
 \wd 217, 632, 633
 \write 146, 524, 527
 tex commands:
 \tex_above:D 286
 \tex_abovedisplayshortskip:D .. 287
 \tex_abovedisplayskip:D 288
 \tex_abovewithdelims:D 289
 \tex_accent:D 290
 \tex_adjdemerits:D 291
 \tex_advance:D .. 292, 5435, 5438,
 5441, 5444, 5451, 5454, 5457, 5460,
 10055, 10059, 10066, 10070, 10440,
 10443, 10448, 10451, 10584, 10587,
 10592, 10595, 17817, 17824, 17827,
 18221, 18223, 18256, 18258, 19461
 \tex_afterassignment:D
 293, 7688, 18162, 18205
 \tex_aftergroup:D 294, 1436
 \tex_atop:D 295
 \tex_atopwithdelims:D 296
 \tex_badness:D 297
 \tex_baselineskip:D 298
 \tex_batchmode:D 299
 \tex_begingroup:D 300, 1250, 1431
 \tex_belowdisplayshortskip:D .. 301
 \tex_belowdisplayskip:D 302
 \tex_binoppenalty:D 303
 \tex_botmark:D 304
 \tex_box:D ... 305, 21792, 21795, 21817
 \tex_boxmaxdepth:D 306
 \tex_brokenpenalty:D 307
 \tex_catcode:D
 308, 2747, 3275, 7154, 7156
 \tex_char:D 309
 \tex_chardef:D 295, 310,
 1424, 1453, 1455, 2022, 2023, 5397,
 6265, 6268, 7612, 9149, 9303, 24079
 \tex_cleaders:D 311
 \tex_closein:D 312, 9157
 \tex_closeout:D 313, 9312
 \tex_clubpenalty:D 314
 \tex_copy:D .. 315, 21784, 21787, 21818
 \tex_count:D 316, 9097, 9099,
 9264, 9266, 17683, 17699, 17707, 17708
 \tex_countdef:D 317
 \tex_cr:D 318
 \tex_crcr:D 319
 \tex_csname:D 320, 1418
 \tex_day:D 321, 6524
 \tex_deadcycles:D 322
 \tex_def:D 323, 544,
 799, 800, 801, 1437, 1439, 1441,
 1442, 1459, 1461, 1462, 1463, 1465,
 1466, 1467, 1469, 1470, 1471, 9958
 \tex_defaultthyphenchar:D 324
 \tex_defaultskewchar:D 325
 \tex_delcode:D 326
 \tex_delimiter:D 327
 \tex_delimiterfactor:D 328
 \tex_delimitershortfall:D 329
 \tex_dimen:D 330
 \tex_dimendef:D 331
 \tex_discretionary:D 332
 \tex_displayindent:D 333
 \tex_displaylimits:D 334
 \tex_displaystyle:D 335
 \tex_displaywidowpenalty:D 336
 \tex_displaywidth:D 337
 \tex_divide:D 338, 17676, 19462
 \tex_doublehyphendemerits:D ... 339
 \tex_dp:D 340, 21803
 \tex_dump:D 341
 \tex_edef:D
 342, 1251, 1252, 1268, 1460,
 1464, 1468, 1472, 9504, 9561, 27152
 \tex_else:D 343, 1254, 1280, 1405, 1456
 \tex_emergencystretch:D 344
 \tex_end:D . 345, 1240, 1374, 2133, 8433
 \tex_endcsname:D 346, 1419
 \tex_endgroup:D 347, 1237, 1276, 1432
 \tex_endinput:D 348, 8444, 24014
 \tex_endlinechar:D
 250, 251, 265, 349, 3248,
 3249, 3250, 3306, 9200, 9202, 9203
 \tex_eqno:D 350
 \tex_errhelp:D 351, 8294
 \tex_errmessage:D 352, 2125, 8325
 \tex_errorcontextlines:D
 .. 353, 3948, 8320, 8352, 8517, 21883
 \tex_errorstopmode:D 354
 \tex_escapechar:D 355, 2441, 9386,
 9422, 9428, 18127, 18189, 18190, 18514
 \tex_everycr:D 356

- `\tex_everydisplay:D` 357, 1241
- `\tex_everyhbox:D` 358
- `\tex_everyjob:D` 359, 1375, 6511, 6513, 9658, 9660, 9694, 9696
- `\tex_everymath:D` 360, 1242
- `\tex_everypar:D` 361
- `\tex_everyvbox:D` 362
- `\tex_exhyphenpenalty:D` 363
- `\tex_expandafter:D` 364, 804, 1255, 1269, 1271, 1272, 1420
- `\tex_fam:D` 365
- `\tex_fi:D` 366, 805, 1249, 1273, 1275, 1284, 1285, 1286, 1344, 1346, 1347, 1351, 1365, 1370, 1385, 1393, 1398, 1406, 1458
- `\tex_finalhyphenemerits:D` 367
- `\tex_firstmark:D` 368
- `\tex_floatingpenalty:D` 369
- `\tex_font:D` 370, 17569
- `\tex_fontdimen:D` 371, 17573, 17574, 17597, 17625
- `\tex_fontname:D` 372
- `\tex_futurelet:D` 373, 7683, 7685, 18135, 18193
- `\tex_gdef:D` 374, 1473, 1476, 1480, 1484
- `\tex_global:D` 271, 276, 278, 375, 548, 806, 808, 1271, 2207, 2214, 5374, 5380, 5384, 5406, 5419, 5441, 5444, 5457, 5460, 5470, 6268, 7418, 7420, 7430, 7685, 9149, 9303, 10009, 10016, 10042, 10051, 10059, 10070, 10399, 10405, 10431, 10436, 10443, 10451, 10533, 10541, 10573, 10580, 10587, 10595, 17569, 21787, 21795, 21849, 21904, 21918, 21933, 21955, 22002, 22016, 22030, 22045, 22067, 24079
- `\tex_globaldefs:D` 376
- `\tex_halign:D` 377
- `\tex_hangafter:D` 378
- `\tex_hangindent:D` 379
- `\tex_hbadness:D` 380
- `\tex_hbox:D` 381, 21894, 21898, 21904, 21912, 21918, 21926, 21933, 21948, 21955, 21963, 21968, 22578
- `\tex_hfil:D` 382
- `\tex_hfill:D` 383
- `\tex_hfilneg:D` 384
- `\tex_hfuzz:D` 385
- `\tex_hoffset:D` 386, 1387
- `\tex_holdinginserts:D` 387
- `\tex_hrule:D` 388
- `\tex_hsize:D` 389, 22494, 22496, 22497, 22540, 22542, 22543
- `\tex_hskip:D` 390, 10485
- `\tex_hss:D` 391, 21972, 21974, 22366, 22375
- `\tex_ht:D` 392, 21802
- `\tex_hyphen:D` 285, 1243
- `\tex_hyphenation:D` 393
- `\tex_hyphenchar:D` 394, 17571, 17572, 17573, 17576
- `\tex_hyphenpenalty:D` 395
- `\tex_if:D` 113, 396, 1408, 1409
- `\tex_ifcase:D` 397, 5265
- `\tex_ifcat:D` 398, 1410
- `\tex_ifdim:D` 399, 9994
- `\tex_ifeof:D` 400, 9177
- `\tex_iffalse:D` 401, 1403
- `\tex_ifhbox:D` 402, 21829
- `\tex_ifhmode:D` 403, 1413
- `\tex_ifinner:D` 404, 1415
- `\tex_ifmmode:D` 405, 1412
- `\tex_ifnum:D` 406, 1345, 1434
- `\tex_ifodd:D` 407, 1531, 1639, 1716, 5264, 6238, 6239
- `\tex_iftrue:D` 408, 1402
- `\tex_ifvbox:D` 409, 21830
- `\tex_ifvmode:D` 410, 1414
- `\tex_ifvoid:D` 411, 21831
- `\tex_ifx:D` 412, 1253, 1270, 1411
- `\tex_ignorespaces:D` 413
- `\tex_immediate:D` 414, 2142, 2144, 9305, 9312, 9353, 26293
- `\tex_indent:D` 415, 23589
- `\tex_input:D` 416, 1244, 1376, 9809, 24379, 24404
- `\tex_inputlineno:D` ... 417, 2140, 8268
- `\tex_insert:D` 418
- `\tex_insertpenalties:D` 419
- `\tex_interlinepenalty:D` 420
- `\tex_italiccorrection:D` 284, 1245, 1388
- `\tex_jobname:D` 421, 6514, 6518, 9661, 9679, 9680
- `\tex_kern:D` 422, 22122, 22364, 22373, 22824, 22829, 22902, 22903, 23193, 23194, 23602, 23604, 23648, 23650, 23730
- `\tex_language:D` 423, 1377
- `\tex_lastbox:D` 424, 21846, 21849
- `\tex_lastkern:D` 425
- `\tex_lastpenalty:D` 426
- `\tex_lastskip:D` 427
- `\tex_lccode:D` 428, 3131, 3132, 3133, 4269, 4270, 4292, 4293, 7230, 7232, 18108, 18118, 18187, 18189, 18192, 18222, 21041, 21093

<code>\tex_leaders:D</code>	429	<code>\tex_mathcode:D</code>	451, 7224, 7226
<code>\tex_left:D</code>	430, 1395	<code>\tex_mathinner:D</code>	452
<code>\tex_lefthyphenmin:D</code>	431	<code>\tex_mathop:D</code>	453, 1378
<code>\tex_leftskip:D</code>	432	<code>\tex_mathopen:D</code>	454
<code>\tex_leqno:D</code>	433	<code>\tex_mathord:D</code>	455
<code>\tex_let:D</code> . 272, 276, 278, 303, 433, 434, 806, 808, 1240, 1241, 1242, 1243, 1244, 1245, 1246, 1248, 1271, 1277, 1279, 1283, 1288, 1289, 1290, 1291, 1292, 1293, 1294, 1295, 1297, 1298, 1299, 1300, 1301, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1315, 1316, 1317, 1318, 1320, 1321, 1323, 1324, 1325, 1327, 1328, 1329, 1330, 1331, 1333, 1334, 1335, 1336, 1337, 1338, 1339, 1340, 1341, 1342, 1343, 1349, 1350, 1355, 1356, 1357, 1358, 1359, 1360, 1361, 1362, 1363, 1364, 1367, 1368, 1369, 1374, 1375, 1376, 1377, 1378, 1379, 1380, 1381, 1382, 1383, 1384, 1387, 1388, 1389, 1390, 1391, 1392, 1395, 1396, 1397, 1402, 1403, 1404, 1405, 1406, 1407, 1408, 1409, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1417, 1418, 1419, 1420, 1421, 1422, 1423, 1425, 1426, 1427, 1428, 1429, 1430, 1431, 1432, 1434, 1435, 1436, 1452, 1459, 1460, 1473, 1474, 1762, 1767, 1783, 1788, 2203, 6155, 7418, 7420, 7430, 18109, 18119		<code>\tex_mathpunct:D</code>	456
<code>\tex_limits:D</code>	435	<code>\tex_mathrel:D</code>	457
<code>\tex_linepenalty:D</code>	436	<code>\tex_mathsurround:D</code>	458
<code>\tex_lineskip:D</code>	437	<code>\tex_maxdeadcycles:D</code>	459
<code>\tex_lineskiplimit:D</code>	438	<code>\tex_maxdepth:D</code>	460
<code>\tex_long:D</code>	439, 799, 800, 801, 1437, 1439, 1442, 1461, 1462, 1463, 1464, 1465, 1467, 1469, 1470, 1471, 1472, 1476, 1478, 1484, 1486	<code>\tex_meaning:D</code>	461, 1252, 1269, 1425, 1426
<code>\tex_looseness:D</code>	440	<code>\tex_medmuskip:D</code>	462
<code>\tex_lower:D</code>	441, 21828	<code>\tex_message:D</code>	463
<code>\tex_lowercase:D</code>	442, 1010, 3134, 3282, 3299, 4271, 4294, 7261, 7383, 8306, 18109, 18119, 18188, 21042, 21094, 25627, 27223	<code>\tex_middle:D</code>	1396
<code>\tex_mag:D</code>	443	<code>\tex_mkern:D</code>	464
<code>\tex_mark:D</code>	444	<code>\tex_month:D</code>	465, 1379, 6525
<code>\tex_mathaccent:D</code>	445	<code>\tex_moveleft:D</code>	466, 21822
<code>\tex_mathbin:D</code>	446	<code>\tex_moveright:D</code>	467, 21824
<code>\tex_mathchar:D</code>	447	<code>\tex_mskip:D</code>	468
<code>\tex_mathchardef:D</code>	295, 448, 1457, 5400, 5401	<code>\tex_multiply:D</code>	469
<code>\tex_mathchoice:D</code>	449	<code>\tex_muskip:D</code>	470
<code>\tex_mathclose:D</code>	450	<code>\tex_muskipdef:D</code>	471
		<code>\tex_newlinechar:D</code> 472, 2124, 3250, 3278, 3281, 3946, 8318, 8515, 9352	
		<code>\tex_noalign:D</code>	473
		<code>\tex_noboundary:D</code>	474
		<code>\tex_noexpand:D</code>	475, 1421
		<code>\tex_noindent:D</code>	476
		<code>\tex_nolimits:D</code>	477
		<code>\tex_nonscript:D</code>	478
		<code>\tex_nonstopmode:D</code>	479
		<code>\tex_nulldelimiterspace:D</code>	480
		<code>\tex_nullfont:D</code>	481, 7641
		<code>\tex_number:D</code>	482, 5261, 22429
		<code>\tex_omit:D</code>	483
		<code>\tex_openin:D</code>	484, 9151
		<code>\tex_openout:D</code>	485, 9305
		<code>\tex_or:D</code>	486, 1404
		<code>\tex_outer:D</code>	487, 1380, 27152
		<code>\tex_output:D</code>	488
		<code>\tex_outputpenalty:D</code>	489
		<code>\tex_over:D</code>	490, 1381
		<code>\tex_overfullrule:D</code>	491
		<code>\tex_overline:D</code>	492
		<code>\tex_overwithdelims:D</code>	493
		<code>\tex_pagedepth:D</code>	494
		<code>\tex_pagefilllstretch:D</code>	495
		<code>\tex_pagefillstretch:D</code>	496
		<code>\tex_pagefilstretch:D</code>	497
		<code>\tex_pagegoal:D</code>	498
		<code>\tex_pageshrink:D</code>	499
		<code>\tex_pagestretch:D</code>	500
		<code>\tex_pagetotal:D</code>	501

<code>\tex_par:D</code>	502	<code>\tex_span:D</code>	546
<code>\tex_parfillskip:D</code>	503	<code>\tex_special:D</code>	547, 25697
<code>\tex_parindent:D</code>	504	<code>\tex_splitbotmark:D</code>	548
<code>\tex_parshape:D</code>	505	<code>\tex_splitfirstmark:D</code>	549
<code>\tex_parskip:D</code>	506	<code>\tex_splitmaxdepth:D</code>	550
<code>\tex_patterns:D</code>	507	<code>\tex_splittopskip:D</code>	551
<code>\tex_pausing:D</code>	508	<code>\tex_string:D</code> ..	552, 1251, 1255, 1428
<code>\tex_penalty:D</code>	509	<code>\tex_tabskip:D</code>	553
<code>\tex_postdisplaypenalty:D</code>	510	<code>\tex_textfont:D</code>	554
<code>\tex_predisplaypenalty:D</code>	511	<code>\tex_textstyle:D</code>	555
<code>\tex_predisplaysize:D</code>	512	<code>\tex_the:D</code>	251,
<code>\tex_pretolerance:D</code>	513		288, 329, 556, 626, 632, 633, 1684,
<code>\tex_prevdepth:D</code>	514		2140, 2427, 2507, 2511, 5473, 5475,
<code>\tex_prevgraf:D</code>	515		6513, 7156, 7226, 7232, 7238, 7244,
<code>\tex_radical:D</code>	516		9660, 9696, 10277, 10278, 10279,
<code>\tex_raise:D</code>	517, 21826		10333, 10335, 10482, 10484, 10607,
<code>\tex_read:D</code>	518, 9195		12574, 13064, 17576, 17774, 17806,
<code>\tex_relax:D</code>			17854, 17855, 17877, 17878, 18346,
	288, 302, 519, 592, 1430, 5263, 9996		18456, 18515, 18534, 18540, 18543,
<code>\tex_relpenalty:D</code>	520		18547, 21867, 24214, 24260, 26316
<code>\tex_right:D</code>	521, 1397	<code>\tex_thickmuskip:D</code>	557
<code>\tex_righthypphenmin:D</code>	522	<code>\tex_thinmuskip:D</code>	558
<code>\tex_rightskip:D</code>	523	<code>\tex_time:D</code>	559, 6521, 6523
<code>\tex_romannumeral:D</code>		<code>\tex_toks:D</code>	560, 17747, 17774,
	302, 312, 312, 335, 524,		17806, 17843, 17854, 17855, 17877,
	1423, 1435, 1695, 2027, 7273, 11575		17878, 17886, 17896, 17906, 18170,
<code>\tex_scriptfont:D</code>	525		18188, 18346, 18515, 18518, 18525,
<code>\tex_scriptscriptfont:D</code>	526		18533, 18534, 18539, 18540, 18543,
<code>\tex_scriptscriptstyle:D</code>	527		18547, 24214, 24225, 24226, 24227
<code>\tex_scriptspace:D</code>	528	<code>\tex_toksdef:D</code>	561, 18014
<code>\tex_scriptstyle:D</code>	529	<code>\tex_tolerance:D</code>	562
<code>\tex_scrollmode:D</code>	530	<code>\tex_topmark:D</code>	563
<code>\tex_setbox:D</code>	531, 21784,	<code>\tex_topskip:D</code>	564
	21787, 21792, 21795, 21846, 21849,	<code>\tex_tracingcommands:D</code>	565
	21898, 21904, 21912, 21918, 21926,	<code>\tex_tracinglostchars:D</code>	566
	21933, 21948, 21955, 21996, 22002,	<code>\tex_tracingmacros:D</code>	567
	22010, 22016, 22024, 22030, 22038,	<code>\tex_tracingonline:D</code>	568, 21882
	22045, 22060, 22067, 22079, 22578	<code>\tex_tracingoutput:D</code>	569
<code>\tex_setlanguage:D</code>	532	<code>\tex_tracingpages:D</code>	570
<code>\tex_sfcode:D</code>	533, 7242, 7244	<code>\tex_tracingparagraphs:D</code>	571
<code>\tex_shipout:D</code>	534, 1248, 1272	<code>\tex_tracingrestores:D</code>	572
<code>\tex_show:D</code>	535	<code>\tex_tracingstats:D</code>	573
<code>\tex_showbox:D</code>	536, 21884	<code>\tex_uccode:D</code>	574, 7236, 7238
<code>\tex_showboxbreadth:D</code> ...	537, 21880	<code>\tex_uchyph:D</code>	575
<code>\tex_showboxdepth:D</code>	538, 21881	<code>\tex_undefined:D</code>	
<code>\tex_showlists:D</code>	539		278, 479, 808, 1277, 1349,
<code>\tex_showthe:D</code>	540		1350, 1367, 1368, 1369, 2220, 2228,
<code>\tex_skewchar:D</code>	541		6171, 10991, 11005, 11038, 11059,
<code>\tex_skip:D</code>	542, 18225,		17686, 18109, 18119, 18197, 18297
	18254, 18273, 18331, 18347, 18353	<code>\tex_underline:D</code>	576, 1246
<code>\tex_skipdef:D</code>	543	<code>\tex_unhbox:D</code>	577, 21976
<code>\tex_space:D</code>	283	<code>\tex_unhcopy:D</code>	578, 21975
<code>\tex_spacefactor:D</code>	544	<code>\tex_unkern:D</code>	579
<code>\tex_spaceskip:D</code>	545	<code>\tex_unpenalty:D</code>	580

- `\tex_unskip:D` 581
- `\tex_unvbox:D` 582, 22074
- `\tex_unvcopy:D` 583, 22073
- `\tex_uppercase:D` 584, 27225
- `\tex_vadjust:D` 585
- `\tex_valign:D` 586
- `\tex_vbadness:D` 587
- `\tex_vbox:D` 588, 21980,
21985, 21990, 21996, 22002, 22024,
22030, 22038, 22045, 22060, 22067
- `\tex_vcenter:D` 589, 1382
- `\tex_vfil:D` 590
- `\tex_vfill:D` 591
- `\tex_vfilneg:D` 592
- `\tex_vfuzz:D` 593
- `\tex_voffset:D` 594, 1389
- `\tex_vrule:D` 595, 23045, 23100
- `\tex_vsize:D` 596
- `\tex_vskip:D` 597, 10495
- `\tex_vsplit:D` 598, 22079
- `\tex_vss:D` 599
- `\tex_vtop:D` .. 600, 21982, 22010, 22016
- `\tex_wd:D` 601, 21804
- `\tex_widowpenalty:D` 602
- `\tex_write:D`
... 603, 2142, 2144, 9333, 9336, 9353
- `\tex_xdef:D` 604, 1474, 1478, 1482, 1486
- `\tex_xleaders:D` 605
- `\tex_xspaceskip:D` 606
- `\tex_year:D` 607, 6526
- `\textdir` 989
- `\textfont` 554
- `\textstyle` 555
- `\texttt` 14091
- `\TeXeTstate` 665
- `\tfont` 1222
- `\TH` 25310
- `\th` 25310
- `\the` 67, 211, 212, 213,
214, 215, 216, 217, 218, 219, 220, 556
- `\thickmuskip` 557
- `\thinmuskip` 558
- thousand commands:
 - `\c_one_thousand` 87, 6132
 - `\c_ten_thousand` 87, 6132
- `\time` 559
- `\tiny` 23038
- tl commands:
 - `\c_empty_tl` ... 46, 424, 449, 3059,
3077, 3079, 3129, 3420, 5842, 5848,
6576, 6587, 10711, 25427, 25466, 25485
 - `\l_my_tl` 202, 208
 - `\c_noalue_tl` 38, 47, 3130, 3503
 - `\c_space_tl`
... 47, 3139, 3729, 4563, 7022, 7031,
8272, 9809, 17336, 23553, 24058,
24060, 24550, 24610, 25715, 25746,
26224, 26229, 26232, 26439, 26442,
26444, 26445, 26446, 26447, 26948
 - `\tl_analysis_map_inline:Nn`
..... 201, 18372, 19929
 - `\tl_analysis_map_inline:nn`
..... 201, 18372, 20525
 - `\tl_analysis_show:N`
..... 201, 18399, 18496, 18498
 - `\tl_analysis_show:n`
..... 201, 18399, 18500, 18502
 - `\tl_build_begin:N` 249, 250,
250, 250, 858, 1004, 19092, 19592,
19955, 20048, 20822, 25377, 25394
 - `\tl_build_clear:N` 249, 25394
 - `\tl_build_end:N` 249,
250, 858, 1004, 1005, 19122, 19130,
19602, 20010, 20067, 20856, 25473
 - `\tl_build_gbegin:N`
.... 249, 250, 250, 250, 25377, 25395
 - `\tl_build_gclear:N` 249, 25394
 - `\tl_build_gend:N` 250, 25473
 - `\tl_build_get:N` 250
 - `\tl_build_get:NN` 250, 25459
 - `\tl_build_gput_left:Nn` ... 250, 25440
 - `\tl_build_gput_right:Nn` ... 250, 25396
 - `\tl_build_put_left:Nn` ... 250, 25440
 - `\tl_build_put_right:Nn` ... 250, 886,
1006, 19099, 19117, 19125, 19129,
19179, 19182, 19215, 19229, 19233,
19360, 19374, 19415, 19440, 19449,
19459, 19491, 19504, 19508, 19589,
19595, 19601, 19605, 19648, 19915,
19931, 19949, 20018, 20063, 20076,
20841, 20880, 20912, 20975, 20978,
20994, 21042, 21058, 21094, 25396
 - `\tl_case:Nn` 39, 3519
 - `\tl_case:nn` 377
 - `\tl_case:nn(TF)` 418
 - `\tl_case:Nnn` 27218, 27220
 - `\tl_case:NnTF`
... 39, 3519, 3524, 3529, 27219, 27221
 - `\l_tl_case_change_accents_tl` ...
..... 246, 24725, 25316
 - `\l_tl_case_change_exclude_tl` ...
..... 246, 246, 24747, 25363
 - `\l_tl_case_change_math_tl`
..... 245, 24560, 25358
 - `\tl_clear:N`
... 34, 35, 3076, 3083, 6628, 6629,
9470, 9471, 9474, 9483, 9592, 9595,

- 9646, 10839, 11333, 11405, 18789,
- 20824, 23988, 25476, 26256, 26264,
- 26270, 26382, 26388, 26468, 26474
- \tl_clear_new:N . 35, 3082, 6632, 6633
- \tl_concat:NNN 35, 3109, 3988
- \tl_const:Nn . . 34, 973, 3062, 3129,
- 3137, 3139, 3228, 4724, 6626, 7385,
- 7414, 7436, 7878, 7916, 8191, 8192,
- 8236, 8241, 8243, 8245, 8247, 8249,
- 8254, 8255, 8262, 9387, 9393, 10775,
- 10776, 10777, 10778, 10779, 10780,
- 10781, 11598, 11599, 11600, 11601,
- 11602, 11610, 11692, 13840, 15117,
- 15562, 15563, 15564, 15565, 15566,
- 15567, 15568, 15569, 15570, 18486,
- 18559, 21127, 23473, 23493, 23505,
- 23536, 23567, 23568, 23569, 24189,
- 25048, 25049, 25050, 25071, 25072,
- 25076, 25077, 25078, 25079, 25080,
- 25089, 25099, 25136, 25149, 25270,
- 25293, 25295, 25313, 25314, 25627
- \tl_count:N 24, 39, 42, 42, 3612
- \tl_count:n
- 24, 39, 42, 42, 320, 385, 411, 595,
- 1884, 1888, 2269, 2319, 3612, 3920,
- 13748, 13769, 19846, 25494, 25514
- \tl_count_tokens:n 244, 24345
- \tl_gclear:N . . 34, 790, 3076, 3085,
- 6630, 6631, 17768, 25481, 26986, 27022
- \tl_gclear_new:N . 35, 3082, 6634, 6635
- \tl_gconcat:NNN 35, 3109, 3989
- \tl_gput_left:Nn 35, 3164
- \tl_gput_right:Nn
- 35, 1733, 1734, 1760,
- 1765, 1781, 1786, 3196, 4706, 4848
- \tl_gremove_all:Nn 36, 3408
- \tl_gremove_once:Nn 36, 3402
- \tl_greplace_all:Nnn . 36, 3333, 3411
- \tl_greplace_once:Nnn 36, 3333, 3405
- \tl_greverse:N 42, 3759
- .tl_gset:N 169, 11201
- \tl_gset:Nn 35, 66, 352, 1007, 3124,
- 3140, 3232, 3336, 3340, 3650, 3762,
- 4019, 4023, 4763, 4768, 4780, 4817,
- 4834, 4874, 4900, 4978, 5010, 5047,
- 5053, 6647, 6674, 6693, 6736, 6772,
- 6821, 6860, 7913, 7981, 8010, 8046,
- 8054, 8077, 9952, 13838, 17764,
- 18264, 18794, 19910, 24150, 24160,
- 24170, 24364, 24391, 25480, 26945
- \tl_gset_eq:NN 35, 3079,
- 3088, 3985, 4747, 4748, 4749, 4750,
- 6260, 6640, 6641, 6642, 6643, 7901,
- 7902, 7903, 7904, 13845, 17758, 21122
- \tl_gset_from_file:Nnn . . . 247, 24361
- \tl_gset_from_file_x:Nnn . 248, 24388
- \tl_gset_rescan:Nnn 37, 3229
- .tl_gset_x:N 169, 11201
- \tl_gsort:Nn 43, 3691, 17756
- \tl_gtrim_spaces:N 43, 3641
- \tl_head:N 44, 3765
- \tl_head:n 44, 44, 366,
- 367, 1008, 3765, 23517, 23549, 25511
- \tl_head:w
- . 44, 368, 369, 3765, 3805, 3822, 3844
- \tl_if_blank:nTF
- 37, 44, 44, 44, 3414,
- 3792, 3968, 3995, 7030, 7109, 9673,
- 10837, 11329, 11344, 11484, 17918,
- 23469, 23481, 23509, 23515, 23540,
- 23547, 24936, 24956, 25006, 25493
- \tl_if_blank_p:n 37, 3414
- \tl_if_empty:N 4072, 4074, 6875, 6877
- \tl_if_empty:NTF
- 38, 1578, 3418, 9487,
- 9576, 9649, 10822, 10847, 11349,
- 20879, 26259, 26300, 26308, 26409,
- 26413, 26440, 26455, 26489, 26948
- \tl_if_empty:nTF 38, 357, 358,
- 359, 449, 458, 1693, 1913, 1995,
- 3237, 3347, 3428, 3439, 3484, 3883,
- 3904, 4030, 4626, 4633, 4650, 4783,
- 6581, 6600, 6608, 6611, 6888, 6921,
- 7629, 7783, 7947, 8277, 8512, 8604,
- 8619, 8739, 8743, 8784, 9001, 9002,
- 9009, 9016, 9022, 9029, 9481, 9866,
- 9869, 10888, 10987, 11566, 12320,
- 13169, 17320, 17419, 18490, 18491,
- 21701, 23534, 24316, 25660, 27042
- \tl_if_empty_p:N 38, 3418, 26296, 26452
- \tl_if_empty_p:n 38, 3428, 3439
- \tl_if_eq:NN 377
- \tl_if_eq:nn(TF) 107, 107
- \tl_if_eq:NNTF
- 38, 39, 60, 398, 3450, 3543,
- 4886, 8097, 8585, 8639, 23111, 23114
- \tl_if_eq:nnTF
- 38, 69, 69, 398, 3460, 6909
- \tl_if_eq_p:NN 38, 3450
- \tl_if_exist:N 4068, 4070
- \tl_if_exist:NTF 35, 3083,
- 3085, 3127, 3605, 18401, 24417, 24425
- \tl_if_exist_p:N 35, 3127
- \tl_if_head_eq_catcode:nN 368
- \tl_if_head_eq_catcode:nNTF
- 45, 3798, 24478, 24478
- \tl_if_head_eq_catcode_p:nN 45, 3798
- \tl_if_head_eq_charcode:nN 368

- \tl_if_head_eq_charcode:nNTF [45](#), [3798](#)
- \tl_if_head_eq_charcode_p:nN [45](#), [3798](#)
- \tl_if_head_eq_meaning:nN [369](#)
- \tl_if_head_eq_meaning:nNTF [45](#), [3798](#)
- \tl_if_head_eq_meaning_p:nN
..... [45](#), [3798](#), [19845](#)
- \tl_if_head_is_group:nTF [45](#), [3705](#),
[3825](#), [3860](#), [3886](#), [6606](#), [24505](#), [24587](#)
- \tl_if_head_is_group_p:n ... [45](#), [3886](#)
- \tl_if_head_is_N_type:n [368](#)
- \tl_if_head_is_N_type:nTF
..... [45](#), [3702](#), [3802](#), [3819](#), [3836](#),
[3869](#), [24313](#), [24502](#), [24584](#), [24838](#),
[24872](#), [24965](#), [25015](#), [25331](#), [25592](#)
- \tl_if_head_is_N_type_p:n .. [45](#), [3869](#)
- \tl_if_head_is_space:nTF
.. [45](#), [3897](#), [4557](#), [25574](#), [25583](#), [25710](#)
- \tl_if_head_is_space_p:n ... [45](#), [3897](#)
- \tl_if_in:Nn [459](#)
- \tl_if_in:nn [358](#)
- \tl_if_in:NnTF
[38](#), [3369](#), [3475](#), [3475](#), [3476](#), [4700](#), [9734](#)
- \tl_if_in:nnTF
..... [38](#), [358](#), [377](#), [3289](#), [3353](#),
[3355](#), [3475](#), [3476](#), [3477](#), [3480](#), [4123](#),
[4131](#), [8498](#), [8500](#), [18676](#), [22917](#), [25658](#)
- \tl_if_novalue:nTF [38](#), [3490](#)
- \tl_if_novalue_p:n [38](#), [3490](#)
- \tl_if_single:n [359](#)
- \tl_if_single:NTF [39](#), [3504](#), [3505](#), [3506](#)
- \tl_if_single:nTF
..... [39](#), [3505](#), [3506](#), [3507](#), [3508](#)
- \tl_if_single_p:N [39](#), [3504](#)
- \tl_if_single_p:n [39](#), [3504](#), [3508](#)
- \tl_if_single_token:nTF .. [244](#), [24311](#)
- \tl_if_single_token_p:n .. [244](#), [24311](#)
- \tl_item:Nn [46](#), [3909](#)
- \tl_item:nn [46](#), [1007](#), [3909](#), [25494](#)
- \tl_log:N [46](#), [3932](#)
- \tl_log:n
... [46](#), [324](#), [324](#), [667](#), [2419](#), [2435](#),
[3934](#), [3956](#), [6182](#), [6288](#), [13870](#), [24055](#)
- \tl_lower_case:n .. [57](#), [116](#), [244](#), [24479](#)
- \tl_lower_case:nn [244](#), [24479](#)
- \tl_map_break:
..... [40](#), [213](#), [813](#), [3556](#), [3562](#),
[3574](#), [3584](#), [3591](#), [3596](#), [18395](#), [18396](#)
- \tl_map_break:n
..... [40](#), [41](#), [3596](#), [9749](#), [17763](#)
- \tl_map_function:NN [39](#), [39](#),
[39](#), [242](#), [242](#), [3552](#), [3620](#), [9728](#), [19917](#)
- \tl_map_function:nN [39](#), [39](#),
[40](#), [2313](#), [3049](#), [3552](#), [3615](#), [4786](#), [19199](#)
- \tl_map_inline:Nn
..... [39](#), [40](#), [3566](#), [9748](#), [17763](#)
- \tl_map_inline:nn [40](#), [40](#), [62](#),
[3566](#), [6557](#), [9390](#), [13222](#), [16234](#), [17714](#)
- \tl_map_variable:NNn [40](#), [3580](#)
- \tl_map_variable:nNn .. [40](#), [361](#), [3580](#)
- \tl_mixed_case:n
..... [57](#), [244](#), [246](#), [250](#), [24479](#)
- \tl_mixed_case:nn [244](#), [24479](#)
- \l_tl_mixed_case_ignore_tl
..... [247](#), [24799](#), [25368](#)
- \l_tl_mixed_change_ignore_tl .. [247](#)
- \tl_new:N [34](#), [35](#),
[118](#), [347](#), [3056](#), [3083](#), [3085](#), [3473](#),
[3474](#), [3958](#), [3959](#), [3960](#), [3961](#), [4696](#),
[4721](#), [4722](#), [6577](#), [6623](#), [6624](#), [7309](#),
[7677](#), [7877](#), [8189](#), [8530](#), [8531](#), [8955](#),
[9090](#), [9235](#), [9257](#), [9364](#), [9366](#), [9379](#),
[9381](#), [9382](#), [9384](#), [9700](#), [9951](#), [10626](#),
[10627](#), [10628](#), [10783](#), [10785](#), [10786](#),
[10789](#), [10790](#), [10794](#), [10795](#), [17642](#),
[18072](#), [18550](#), [18551](#), [18557](#), [18558](#),
[18568](#), [20506](#), [20746](#), [20748](#), [22398](#),
[22423](#), [22424](#), [23033](#), [23280](#), [25316](#),
[25358](#), [25363](#), [25368](#), [26252](#), [26952](#)
- \tl_put_left:Nn [35](#), [3164](#)
- \tl_put_right:Nn
.. [35](#), [1005](#), [1571](#), [3196](#), [4846](#), [7350](#),
[7352](#), [7355](#), [7357](#), [7363](#), [7365](#), [7367](#),
[7369](#), [7370](#), [7372](#), [7374](#), [7376](#), [7378](#),
[9608](#), [9611](#), [9616](#), [11347](#), [18796](#), [20915](#)
- \tl_rand_item:N [248](#), [25491](#)
- \tl_rand_item:n [248](#), [25491](#)
- \tl_range:Nnn [248](#), [25498](#)
- \tl_range:nnn .. [248](#), [249](#), [1007](#), [25498](#)
- \tl_range_braced:Nnn [249](#), [25498](#)
- \tl_range_braced:nnn [248](#), [249](#), [25498](#)
- \tl_range_unbraced:Nnn ... [249](#), [25498](#)
- \tl_range_unbraced:nnn [248](#), [249](#), [25498](#)
- \tl_remove_all:Nn .. [36](#), [36](#), [3408](#), [9733](#)
- \tl_remove_once:Nn [36](#), [3402](#)
- \tl_replace_all:Nnn
..... [36](#), [395](#), [456](#), [3333](#), [3409](#), [4795](#)
- \tl_replace_once:Nnn
..... [36](#), [3333](#), [3403](#), [7393](#)
- \tl_rescan:nn [37](#), [37](#), [3229](#)
- \tl_reverse:N [42](#), [42](#), [3759](#)
- \tl_reverse:n
.... [42](#), [42](#), [42](#), [980](#), [3739](#), [3760](#), [3762](#)
- \tl_reverse_items:n . [42](#), [42](#), [42](#), [3625](#)
- \tl_reverse_tokens:n [244](#), [24321](#)
- .tl_set:N [169](#), [11201](#)
- \tl_set:Nn
[35](#), [36](#), [37](#), [66](#), [169](#), [250](#), [250](#), [336](#),

- [347](#), [352](#), [507](#), [1007](#), [1576](#), [3116](#),
[3140](#), [3230](#), [3260](#), [3319](#), [3334](#), [3338](#),
[3463](#), [3464](#), [3590](#), [3648](#), [3760](#), [3945](#),
[4017](#), [4021](#), [4753](#), [4758](#), [4778](#), [4785](#),
[4789](#), [4800](#), [4815](#), [4826](#), [4872](#), [4882](#),
[4891](#), [4898](#), [4929](#), [4932](#), [4949](#), [4957](#),
[4966](#), [4976](#), [4985](#), [4991](#), [5008](#), [5022](#),
[5044](#), [5050](#), [5139](#), [5739](#), [6645](#), [6672](#),
[6691](#), [6725](#), [6731](#), [6734](#), [6740](#), [6747](#),
[6770](#), [6819](#), [6858](#), [6998](#), [7348](#), [7353](#),
[7695](#), [7910](#), [7975](#), [7991](#), [7992](#), [8000](#),
[8001](#), [8003](#), [8009](#), [8012](#), [8035](#), [8036](#),
[8045](#), [8053](#), [8057](#), [8075](#), [8080](#), [8130](#),
[8514](#), [8542](#), [8625](#), [8953](#), [9141](#), [9295](#),
[9371](#), [9434](#), [9436](#), [9437](#), [9446](#), [9451](#),
[9472](#), [9619](#), [9638](#), [9640](#), [9715](#), [9718](#),
[9719](#), [9979](#), [10319](#), [10635](#), [10654](#),
[10661](#), [10678](#), [10685](#), [10696](#), [10762](#),
[10801](#), [10803](#), [10831](#), [10845](#), [10851](#),
[10854](#), [10863](#), [10864](#), [10867](#), [10969](#),
[11227](#), [11229](#), [11240](#), [11241](#), [11265](#),
[11266](#), [11306](#), [11327](#), [11339](#), [11345](#),
[11407](#), [13836](#), [14165](#), [18674](#), [18687](#),
[18781](#), [19133](#), [19542](#), [19547](#), [19810](#),
[19871](#), [19901](#), [20013](#), [20070](#), [20576](#),
[20585](#), [20669](#), [20701](#), [21020](#), [21299](#),
[21368](#), [21398](#), [21422](#), [22596](#), [22918](#),
[22919](#), [23035](#), [23038](#), [23281](#), [23978](#),
[24148](#), [24158](#), [24168](#), [24362](#), [24387](#),
[24389](#), [24403](#), [25107](#), [25126](#), [25279](#),
[25317](#), [25360](#), [25365](#), [25369](#), [25460](#),
[25475](#), [25712](#), [25724](#), [25775](#), [25778](#),
[25781](#), [25785](#), [25788](#), [26257](#), [26272](#)
\ [tl_set_eq:N](#) [35](#), [437](#), [3077](#), [3088](#), [3984](#), [4743](#),
[4744](#), [4745](#), [4746](#), [6259](#), [6636](#), [6637](#),
[6638](#), [6639](#), [7897](#), [7898](#), [7899](#), [7900](#),
[8588](#), [8596](#), [11401](#), [13844](#), [17756](#), [21117](#)
\ [tl_set_from_file:Nnn](#) ... [247](#), [24361](#)
\ [tl_set_from_file_x:Nnn](#) .. [248](#), [24388](#)
\ [tl_set_rescan:Nnn](#) [37](#), [37](#), [3229](#)
. [tl_set_x:N](#) [169](#), [11201](#)
\ [tl_show:N](#) [46](#), [46](#), [814](#), [3932](#), [4593](#), [18407](#)
\ [tl_show:n](#) [46](#), [46](#), [239](#), [324](#),
[324](#), [372](#), [372](#), [667](#), [972](#), [2415](#), [2432](#),
[3932](#), [3941](#), [4592](#), [6181](#), [6286](#), [7158](#),
[7228](#), [7234](#), [7240](#), [7246](#), [13868](#), [24053](#)
\ [tl_show_analysis:N](#) [18495](#)
\ [tl_show_analysis:n](#) [18495](#)
\ [tl_sort:Nn](#) [43](#), [3691](#), [17756](#)
\ [tl_sort:nN](#) . [43](#), [795](#), [796](#), [3691](#), [17914](#)
\ [tl_tail:N](#) [44](#), [1577](#), [3765](#), [19821](#)
\ [tl_tail:n](#) [44](#), [3765](#)
\ [tl_to_lowercase:n](#) [27222](#)
\ [tl_to_str:N](#) [41](#), [48](#), [147](#),
[374](#), [531](#), [889](#), [3601](#), [4038](#), [4115](#),
[4123](#), [9437](#), [9442](#), [9720](#), [9892](#), [9906](#)
\ [tl_to_str:n](#) [37](#), [37](#), [41](#), [41](#), [48](#), [57](#),
[58](#), [129](#), [129](#), [147](#), [166](#), [173](#), [207](#),
[208](#), [293](#), [308](#), [359](#), [374](#), [380](#), [386](#),
[477](#), [493](#), [494](#), [889](#), [889](#), [1427](#), [1450](#),
[1539](#), [1547](#), [1558](#), [1563](#), [1774](#), [1903](#),
[1971](#), [2766](#), [2780](#), [2783](#), [2790](#), [2794](#),
[2998](#), [3030](#), [3252](#), [3350](#), [3431](#), [3600](#),
[3942](#), [3957](#), [4000](#), [4039](#), [4123](#), [4131](#),
[4266](#), [4288](#), [4312](#), [4319](#), [4373](#), [4380](#),
[4454](#), [4473](#), [4484](#), [4509](#), [4517](#), [4525](#),
[4531](#), [4543](#), [4554](#), [6014](#), [6031](#), [6075](#),
[6188](#), [7518](#), [7522](#), [7552](#), [7553](#), [7586](#),
[7601](#), [7603](#), [7605](#), [7623](#), [7840](#), [7951](#),
[7962](#), [8020](#), [8021](#), [8059](#), [8082](#), [8104](#),
[8105](#), [8396](#), [8397](#), [8677](#), [8678](#), [9372](#),
[9388](#), [9773](#), [9847](#), [10144](#), [10346](#),
[10477](#), [10722](#), [11491](#), [11729](#), [11733](#),
[11750](#), [11971](#), [11972](#), [12576](#), [12577](#),
[12582](#), [12586](#), [17077](#), [17131](#), [17205](#),
[19199](#), [20979](#), [21134](#), [23063](#), [23146](#),
[24029](#), [24058](#), [24060](#), [24064](#), [24066](#),
[24071](#), [24073](#), [24289](#), [24302](#), [24686](#),
[24689](#), [25653](#), [25658](#), [27156](#), [27159](#)
\ [tl_to_uppercase:n](#) [27224](#)
\ [tl_trim_spaces:N](#) [43](#), [3641](#)
\ [tl_trim_spaces:n](#) [43](#), [3641](#), [4807](#), [11491](#)
\ [tl_trim_spaces_apply:nN](#)
... [43](#), [3641](#), [6583](#), [7122](#), [7933](#), [10763](#)
\ [tl_trim_spaces:n](#) [363](#)
\ [tl_upper_case:n](#)
..... [57](#), [116](#), [244](#), [250](#), [24479](#)
\ [tl_upper_case:nn](#) [244](#), [24479](#)
\ [tl_use:N](#) [42](#), [76](#), [156](#),
[160](#), [163](#), [3603](#), [11037](#), [24418](#), [24426](#)
\ [g_tmpa_tl](#) [47](#), [3958](#)
\ [l_tmpa_tl](#)
.... [4](#), [36](#), [47](#), [1251](#), [1253](#), [1270](#), [3960](#)
\ [g_tmpb_tl](#) [47](#), [3958](#)
\ [l_tmpb_tl](#)
.... [47](#), [1252](#), [1253](#), [1268](#), [1270](#), [3960](#)
tl internal commands:
\ [c__tl_accents_lt_tl](#) [994](#), [24931](#), [25041](#)
\ [__tl_act:NNNnn](#) [364](#),
[364](#), [366](#), [980](#), [3693](#), [3744](#), [24326](#), [24349](#)
\ [__tl_act_count_group:nn](#) [24345](#)
\ [__tl_act_count_normal:nN](#) [24345](#)
\ [__tl_act_count_space:n](#) [24345](#)
\ [__tl_act_end:w](#) [3693](#)
\ [__tl_act_end:wn](#) [980](#), [3714](#), [3720](#)
\ [__tl_act_group:nwnNNN](#) [3693](#)

- __tl_act_group_recurse:Nnn [24336](#), [24340](#)
- __tl_act_loop:w [3693](#)
- __tl_act_normal:NwnNNN [3693](#)
- __tl_act_output:n [365](#), [3693](#)
- __tl_act_result:n [364](#), [3698](#), [3720](#), [3735](#), [3736](#), [3737](#), [3738](#)
- __tl_act_reverse [365](#)
- __tl_act_reverse_output:n [3693](#), [3754](#), [3756](#), [3758](#), [24337](#)
- __tl_act_space:wwnNNN [365](#), [3693](#)
- __tl_analysis:n [804](#), [814](#), [18095](#), [18374](#), [18403](#), [18411](#)
- __tl_analysis_a:n [18099](#), [18124](#)
- __tl_analysis_a_bgroup:w [18155](#), [18177](#)
- __tl_analysis_a_cs:ww [18234](#)
- __tl_analysis_a_egroup:w [18157](#), [18177](#)
- __tl_analysis_a_group:nw [18177](#)
- __tl_analysis_a_group_aux:w [18177](#)
- __tl_analysis_a_group_auxii:w [18177](#)
- __tl_analysis_a_group_test:w [18177](#)
- __tl_analysis_a_loop:w [18131](#), [18134](#), [18175](#), [18217](#), [18231](#), [18249](#)
- __tl_analysis_a_safe:N [18156](#), [18198](#), [18234](#)
- __tl_analysis_a_space:w [18154](#), [18160](#)
- __tl_analysis_a_space_test:w [807](#), [18160](#)
- __tl_analysis_a_store: [807](#), [18171](#), [18213](#), [18219](#)
- __tl_analysis_a_type:w [18135](#), [18136](#)
- __tl_analysis_b:n [18100](#), [18262](#)
- __tl_analysis_b_char:Nww [18289](#), [18295](#)
- __tl_analysis_b_cs:Nww [18291](#), [18319](#)
- __tl_analysis_b_cs_test:ww [18319](#)
- __tl_analysis_b_loop:w [812](#), [18262](#), [18365](#), [18370](#)
- __tl_analysis_b_normal:wwN [18275](#), [18340](#)
- __tl_analysis_b_normals:ww [811](#), [812](#), [18272](#), [18275](#), [18316](#), [18326](#)
- __tl_analysis_b_special:w [18278](#), [18337](#)
- __tl_analysis_b_special_char:wN [18337](#)
- __tl_analysis_b_special_space:w [18337](#)
- \l__tl_analysis_char_token [802](#), [807](#), [808](#), [18066](#), [18164](#), [18169](#), [18207](#), [18212](#)
- __tl_analysis_cs_space_count:NN [18079](#), [18248](#), [18322](#)
- __tl_analysis_cs_space_count:w [18079](#)
- __tl_analysis_cs_space_count_-end:w [18079](#)
- __tl_analysis_disable:n [18104](#), [18126](#), [18192](#), [18245](#)
- __tl_analysis_extract_charcode: [18073](#), [18187](#)
- __tl_analysis_extract_charcode_-aux:w [18073](#)
- \l__tl_analysis_index_int [809](#), [810](#), [18069](#), [18129](#), [18132](#), [18170](#), [18188](#), [18225](#), [18228](#), [18254](#), [18256](#), [18343](#)
- __tl_analysis_map_inline_aux:Nn [18372](#)
- __tl_analysis_map_inline_-aux:nnn [18372](#)
- \l__tl_analysis_nesting_int [806](#), [18070](#), [18130](#), [18221](#), [18230](#)
- \l__tl_analysis_normal_int [18068](#), [18128](#), [18173](#), [18215](#), [18226](#), [18229](#), [18246](#), [18255](#), [18260](#)
- \g__tl_analysis_result_tl [813](#), [18072](#), [18264](#), [18394](#), [18417](#)
- __tl_analysis_show: [18405](#), [18413](#), [18415](#)
- __tl_analysis_show_active:n [18430](#), [18459](#)
- __tl_analysis_show_cs:n [18426](#), [18459](#)
- \c__tl_analysis_show_etc_str [815](#), [18479](#), [18481](#), [18486](#)
- __tl_analysis_show_long:nn [18459](#)
- __tl_analysis_show_long_-aux:nnnn [18459](#), [18465](#)
- __tl_analysis_show_loop:wNw [18415](#)
- __tl_analysis_show_normal:n [18433](#), [18439](#)
- __tl_analysis_show_value:N [18444](#), [18468](#)
- \l__tl_analysis_token [802](#), [803](#), [806](#), [808](#), [18066](#), [18076](#), [18135](#), [18139](#), [18142](#), [18145](#), [18193](#), [18197](#), [18212](#)
- \l__tl_analysis_type_int [806](#), [809](#), [18071](#), [18138](#), [18153](#), [18221](#), [18223](#), [18227](#)
- __tl_build_begin:NN [25377](#), [25428](#)
- __tl_build_begin:NNN [1004](#), [25377](#)
- __tl_build_end_loop:NN [25473](#)
- __tl_build_get:NNN [25459](#), [25475](#), [25480](#)

- _tl_build_get:w [25459](#)
- _tl_build_get_end:w [25459](#)
- _tl_build_last:NNn
 [1004](#), [1005](#), [25391](#), [25396](#), [25463](#)
- _tl_build_put:nn [1005](#), [25396](#), [25454](#)
- _tl_build_put:nw [1005](#), [25396](#)
- _tl_build_put_left:NNn [25440](#)
- _tl_case:NnTF
 [3522](#), [3527](#), [3532](#), [3537](#), [3539](#)
- _tl_case:nnTF [3519](#)
- _tl_case:Nw [3519](#)
- _tl_case_end:nw [3519](#)
- _tl_change_case:nnn [24479](#), [24480](#),
 [24481](#), [24482](#), [24483](#), [24484](#), [24485](#)
- _tl_change_case_aux:nnn [24485](#)
- _tl_change_case_char:nN [24485](#)
- _tl_change_case_char_lower:Nnn
 [24485](#)
- _tl_change_case_char_mixed:Nnn
 [24485](#)
- _tl_change_case_char_upper:Nnn
 [24485](#)
- _tl_change_case_char_UTFviii:nn
 [24485](#)
- _tl_change_case_char_UTFviii:nNN
 [24485](#)
- _tl_change_case_char_UTFviii:nnN
 [24679](#), [24681](#), [24683](#), [24684](#)
- _tl_change_case_char_UTFviii:nNNN
 [24485](#)
- _tl_change_case_char_UTFviii:nNNNN
 [24485](#)
- _tl_change_case_char_UTFviii:nNNNNN
 [24665](#), [24682](#)
- _tl_change_case_cs:N [24485](#)
- _tl_change_case_cs:NN [24485](#)
- _tl_change_case_cs:NNn [24485](#)
- _tl_change_case_cs_accents:NN .
 [24485](#)
- _tl_change_case_cs_expand:NN [24485](#)
- _tl_change_case_cs_expand:Nnw .
 [24485](#)
- _tl_change_case_cs_letterlike:Nn
 [24485](#)
- _tl_change_case_cs_letterlike:NnN
 [24485](#)
- _tl_change_case_end:wn [24485](#)
- _tl_change_case_group:nwnn . [24485](#)
- _tl_change_case_group_lower:nnnn
 [24485](#)
- _tl_change_case_group_mixed:nnnn
 [24485](#)
- _tl_change_case_group_upper:nnnn
 [24485](#)
- _tl_change_case_if_expandable:NnTF
 [24485](#),
 [24844](#), [24879](#), [24971](#), [25021](#), [25337](#)
- _tl_change_case_loop:wn [993](#)
- _tl_change_case_loop:wnn
 [985](#), [987](#), [24485](#)
- _tl_change_case_lower_az:Nnw [24856](#)
- _tl_change_case_lower_lt:nNnw .
 [24928](#)
- _tl_change_case_lower_lt:NNw [24928](#)
- _tl_change_case_lower_lt:Nnw [24928](#)
- _tl_change_case_lower_lt:nnw [24928](#)
- _tl_change_case_lower_lt:Nw . [24928](#)
- _tl_change_case_lower_sigma:Nnw
 [24826](#)
- _tl_change_case_lower_sigma:Nw
 [24826](#)
- _tl_change_case_lower_sigma:w .
 [24826](#)
- _tl_change_case_lower_tr:Nnw [24856](#)
- _tl_change_case_lower_tr_auxi:Nw [24856](#)
- _tl_change_case_lower_tr_auxii:Nw [24856](#)
- _tl_change_case_math:NNNnnn ...
 [986](#), [24485](#)
- _tl_change_case_math:NwNNnn . [24485](#)
- _tl_change_case_math_group:nwNNnn
 [24485](#)
- _tl_change_case_math_loop:wNNnn
 [24485](#)
- _tl_change_case_math_space:wNNnn
 [24485](#)
- _tl_change_case_mixed_nl:NNw [25319](#)
- _tl_change_case_mixed_nl:Nnw [25319](#)
- _tl_change_case_mixed_nl:Nw . [25319](#)
- _tl_change_case_mixed_skip:N [24485](#)
- _tl_change_case_mixed_skip:NN .
 [24485](#)
- _tl_change_case_mixed_skip_tidy:Nwn [24485](#)
- _tl_change_case_mixed_switch:w
 [24485](#)
- _tl_change_case_N_type:Nnnn . [24485](#)
- _tl_change_case_N_type:NNNnnn .
 [24485](#)
- _tl_change_case_N_type:Nwnn . [24485](#)
- _tl_change_case_output:nwn ...
 [24485](#),
 [24830](#), [24866](#), [24874](#), [24888](#), [24890](#),
 [24900](#), [24911](#), [24923](#), [24950](#), [24959](#),
 [24987](#), [25009](#), [25038](#), [25325](#), [25351](#)
- _tl_change_case_protect:wNN . [24485](#)

- __tl_change_case_result:n 24498, 24511, 24512, 24514
- __tl_change_case_setup:NN 25290, 25297, 25299
- __tl_change_case_space:wnn .. 24485
- __tl_change_case_upper_az:Nnw 24856
- __tl_change_case_upper_de-alt:Nnw 25035
- __tl_change_case_upper_lt:NNw 24928
- __tl_change_case_upper_lt:Nnw 24928
- __tl_change_case_upper_lt:nw 24928
- __tl_change_case_upper_lt:Nw . 24928
- __tl_change_case_upper_sigma:Nnw 24826
- __tl_change_case_upper_tr:Nnw 24856
- __tl_count:n 362, 3612
- \c_tl_dot_above_tl ... 24987, 25041
- \c_tl_dotless_i_tl 24874, 24888, 24900, 25082
- \c_tl_dotted_I_tl 24923, 25082
- \l_tl_file_name_str 24369, 24370, 24379, 24396, 24397, 24404, 24409
- \c_tl_final_sigma_tl 24840, 24852, 25041
- __tl_from_file_do:w 24361
- __tl_head_auxi:nw 3765
- __tl_head_auxii:n 3765
- \c_tl_I_ogonek_tl 24999, 25082
- \c_tl_i_ogonek_tl 24944, 25082
- __tl_if_blank_p:NNw 3414
- __tl_if_empty_return:n 356, 357, 391, 3415, 3439, 24314, 24318
- __tl_if_head_eq_meaning-normal:nN 3837, 3841
- __tl_if_head_eq_meaning-special:nN 3838, 3850
- __tl_if_head_is_N_type:w . 370, 3869
- __tl_if_head_is_space:w 3897
- __tl_if_novalue:w 3490
- __tl_if_single:nw . 359, 3510, 3518
- __tl_if_single:nTF 3508
- __tl_if_single_p:n 3508
- \l_tl_internal_a_tl .. 349, 372, 3255, 3260, 3319, 3460, 3945, 3951, 24381, 24387, 24403, 24406, 25107, 25109, 25126, 25131, 25279, 25281
- \l_tl_internal_b_tl 3460
- __tl_item:nn 3909
- __tl_item_aux:nn 3909
- __tl_loop:nn ... 25123, 25132, 25163
- __tl_map_function:Nn 361, 3552, 3571
- __tl_map_variable:Nnn 3580
- __tl_range:Nnnn 25498
- __tl_range:nnNn 25498
- __tl_range:nnnNn 25498
- __tl_range:w 1007, 25498
- __tl_range_braced:w 1007, 1008, 25498
- __tl_range_collect:nn .. 1008, 25498
- __tl_range_collect_braced:w ... 1008, 25498
- __tl_range_collect_group:nN . 25498
- __tl_range_collect_group:nn ... 25594, 25603
- __tl_range_collect_N:nN 25498
- __tl_range_collect_space:nw . 25498
- __tl_range_collect_unbraced:w 25498
- __tl_range_items:nnNn 1007
- __tl_range_normalize:nn 25521, 25525, 25605
- __tl_range_skip:w 1008, 25498
- __tl_range_skip_spaces:n 25498
- __tl_range_unbraced:w .. 1007, 25498
- __tl_replace:NnNNNnn 352, 353, 3334, 3336, 3338, 3340, 3345
- __tl_replace_auxi:NnnNNNnn 353, 3345
- __tl_replace_auxii:NnnNnn 353, 354, 3345
- __tl_replace_next:w 352, 354, 355, 3338, 3340, 3345
- __tl_replace_wrap:w 352, 354, 355, 3334, 3336, 3345
- __tl_rescan:w 351, 3229, 3311, 3312, 3327
- \c_tl_rescan_marker_tl 351, 3228, 3247, 3268, 3316, 24375, 24386
- __tl_reverse_group:nn 24321
- __tl_reverse_group_preserve:nn 3739
- __tl_reverse_items:nwNwn 3625
- __tl_reverse_items:wn 3625
- __tl_reverse_normal:nN . 3739, 24327
- __tl_reverse_space:n .. 3739, 24329
- __tl_set_from_file:NNnn 24361
- __tl_set_from_file_x:NNnn ... 24388
- __tl_set_rescan:n .. 349, 3252, 3274
- __tl_set_rescan:NNnn 3229
- __tl_set_rescan:NnTF 3274
- __tl_set_rescan_multi:n 349, 351, 3229, 3284
- __tl_set_rescan_multiple:n ... 350
- __tl_set_rescan_single:nn 350, 3274
- __tl_set_rescan_single_aux:nn 3274
- __tl_show:n 3941
- __tl_show:NN 3932
- __tl_show:w 3941
- \c_tl_std_sigma_tl ... 24851, 25041
- __tl_tmp:n 25046, 25048, 25049, 25052, 25054, 25055, 25056,

- 25058, 25060, 25061, 25062, 25064,
25066, 25067, 25068, 25071, 25072
- _tl_tmp:w 358, 363, 3483,
3484, 3490, 3503, 3653, 3690, 25087,
25094, 25097, 25109, 25113, 25114,
25115, 25116, 25131, 25134, 25265,
25268, 25281, 25284, 25285, 25286
- _tl_trim_spaces:nn 3642, 3645, 3653
- _tl_trim_spaces_auxi:w .. 363, 3653
- _tl_trim_spaces_auxii:w . 363, 3653
- _tl_trim_spaces_auxiii:w 363, 3653
- _tl_trim_spaces_auxiv:w . 363, 3653
- \c_tl_upper_Eszett_tl . 25038, 25041
- \tn 18511
- token commands:
- \c_alignment_token
..... 118, 473, 7416, 7455, 18305
- \c_parameter_token
..... 118, 474, 883, 7416, 7459, 7462
- \g_peek_token ... 122, 122, 7674, 7685
- \l_peek_token 122, 122, 483,
483, 1011, 7674, 7683, 7723, 7735,
7755, 7764, 25636, 25637, 25638, 25641
- \c_space_token ... 30, 45, 47, 118,
251, 368, 474, 3827, 3862, 7416,
7479, 7764, 9541, 18139, 18169,
18311, 18861, 18896, 24404, 25638
- \token_get_arg_spec:N 125, 7838
- \token_get_prefix_spec:N .. 125, 7838
- \token_get_replacement_spec:N ...
..... 125, 7838, 11520
- \token_if_active:NTF 120, 7492
- \token_if_active_p:N 120, 7492
- \token_if_alignment:NTF 119, 119, 7453
- \token_if_alignment_p:N ... 119, 7453
- \token_if_chardef:NTF 121, 7564, 18448
- \token_if_chardef_p:N 121, 7564
- \token_if_cs:NTF 120, 7529, 24617
- \token_if_cs_p:N
120, 7529, 24886, 24978, 25028, 25344
- \token_if_dim_register:NTF
..... 121, 7564, 18450
- \token_if_dim_register_p:N 121, 7564
- \token_if_eq_catcode:NNTF
..... 120, 122, 123, 123, 123, 7502
- \token_if_eq_catcode_p:NN . 120, 7502
- \token_if_eq_charcode:NNTF
..... 120, 123, 123, 123,
124, 7507, 8919, 9541, 9738, 15825,
18896, 18901, 19514, 19712, 19725,
19727, 19765, 19887, 20882, 20962
- \token_if_eq_charcode_p:NN 120, 7507
- \token_if_eq_meaning:NNTF
120, 124, 124, 124, 124, 483, 7497,
9593, 12090, 13097, 13156, 14113,
14115, 14120, 14184, 14368, 16354,
19206, 19520, 19553, 19700, 19723,
19755, 19882, 19885, 20933, 20960,
21002, 21023, 24567, 24595, 24599
- \token_if_eq_meaning_p:NN
..... 120, 7497, 24779
- \token_if_expandable:NTF
..... 120, 7534, 18446, 24775
- \token_if_expandable_p:N .. 120, 7534
- \token_if_group_begin:NTF . 119, 7438
- \token_if_group_begin_p:N . 119, 7438
- \token_if_group_end:NTF ... 119, 7443
- \token_if_group_end_p:N ... 119, 7443
- \token_if_int_register:NTF
..... 121, 7564, 18451
- \token_if_int_register_p:N 121, 7564
- \token_if_letter:N 476
- \token_if_letter:NTF 120, 7482, 24850
- \token_if_letter_p:N 120, 7482
- \token_if_long_macro:NTF .. 120, 7564
- \token_if_long_macro_p:N .. 120, 7564
- \token_if_macro:NTF
... 120, 7512, 7618, 7844, 7853, 7862
- \token_if_macro_p:N 120, 7512
- \token_if_math_subscript:NTF ...
..... 119, 7472
- \token_if_math_subscript_p:N ...
..... 119, 7472
- \token_if_math_superscript:NTF ..
..... 119, 7466
- \token_if_math_superscript_p:N ..
..... 119, 7466
- \token_if_math_toggle:NTF . 119, 7448
- \token_if_math_toggle_p:N . 119, 7448
- \token_if_mathchardef:NTF
..... 121, 7564, 18449
- \token_if_mathchardef_p:N . 121, 7564
- \token_if_muskip_register:NTF ...
..... 121, 7564
- \token_if_muskip_register_p:N ...
..... 121, 7564
- \token_if_other:NTF 120, 7487
- \token_if_other_p:N 120, 7487
- \token_if_parameter:NTF ... 119, 7458
- \token_if_parameter_p:N ... 119, 7458
- \token_if_primitive:NTF ... 122, 7612
- \token_if_primitive_p:N ... 122, 7612
- \token_if_protected_long_macro:NTF 121, 7564
- \token_if_protected_long_macro_p:N 121, 7564, 24781
- \token_if_protected_macro:NTF ...
..... 120, 7564

- \token_if_protected_macro_p:N ... 120, 7564, 24780
 - \token_if_skip_register:NTF ... 121, 7564, 18452
 - \token_if_skip_register_p:N 121, 7564
 - \token_if_space:NTF ... 119, 7477
 - \token_if_space_p:N ... 119, 7477
 - \token_if_toks_register:NTF ... 122, 7564, 18453
 - \token_if_toks_register_p:N 122, 7564
 - \token_new:Nn ... 7869
 - \token_to_meaning:N ... 14, 118, 475, 479, 1425, 1441, 2155, 2780, 7416, 7518, 7585, 7622, 7847, 7856, 7865, 18076, 18442, 18467, 25641
 - \token_to_str:N ... 5, 16, 48, 118, 147, 312, 370, 416, 477, 627, 629, 887, 1427, 1441, 1441, 1599, 1608, 1751, 1774, 1887, 1896, 1917, 1938, 1979, 1984, 1999, 2028, 2029, 2049, 2148, 2155, 2281, 2316, 2323, 2411, 2431, 2444, 2800, 2884, 2899, 2914, 2936, 2960, 2972, 2994, 3015, 3228, 3874, 3890, 3939, 4236, 4703, 5249, 5512, 6293, 6345, 7134, 7416, 7599, 7600, 7605, 7606, 7607, 7608, 7609, 7610, 8181, 9423, 9424, 9425, 9426, 9427, 9433, 10664, 10688, 11728, 11743, 11977, 11978, 12458, 12459, 12488, 12657, 12708, 12740, 12760, 12775, 12787, 12788, 12801, 12802, 12827, 12836, 12838, 12863, 12866, 12891, 12893, 12907, 12923, 12941, 13010, 13020, 13021, 13036, 13037, 13370, 13577, 13875, 16422, 17594, 17619, 18012, 18029, 18084, 18165, 18208, 18238, 18287, 18298, 18300, 18302, 18312, 18348, 18359, 18405, 18441, 18466, 18487, 18807, 18814, 18915, 18919, 19631, 20905, 21143, 21723, 21725, 21889, 22448, 22595, 23216, 23996, 23999, 24205, 24417, 24418, 24425, 24426, 24706, 24709, 24717, 25293, 25295, 25313, 25314, 27156, 27159
 - token internal commands:
 - \c_token_A_int ... 7612, 7649
 - __token_delimit_by_char:w .. 7546
 - __token_delimit_by_count:w .. 7546
 - __token_delimit_by_dimen:w .. 7546
 - __token_delimit_by_macro:w .. 7546
 - __token_delimit_by_muskip:w .. 7546
 - __token_delimit_by_skip:w ... 7546
 - __token_delimit_by_toks:w ... 7546
 - __token_if_macro_p:w ... 7512
 - __token_if_primitive:NNw ... 7612
 - __token_if_primitive:Nw ... 7612
 - __token_if_primitive_loop:N .. 7612
 - __token_if_primitive_nullfont:N ... 7612
 - __token_if_primitive_space:w .. 7612
 - __token_if_primitive_undefined:N ... 7612
 - __token_tmp:w ... 477, 7547, 7556, 7557, 7558, 7559, 7560, 7561, 7562, 7565, 7599, 7600, 7601, 7602, 7604, 7606, 7607, 7608, 7609, 7610
 - \toks ... 560, 7610
 - \toksapp ... 972
 - \toksdef ... 561, 18008
 - \tokspre ... 973
 - \tolerance ... 562
 - \topmark ... 563
 - \topmarks ... 666
 - \topskip ... 564
 - \tpack ... 974
 - \tracingassigns ... 667
 - \tracingcommands ... 565
 - \tracingfonts ... 1020
 - \tracinggroups ... 668
 - \tracingifs ... 669
 - \tracinglostchars ... 566
 - \tracingmacros ... 567
 - \tracingnesting ... 670
 - \tracingonline ... 568
 - \tracingoutput ... 569
 - \tracingpages ... 570
 - \tracingparagraphs ... 571
 - \tracingrestores ... 572
 - \tracingscantokens ... 671
 - \tracingstats ... 573
 - true ... 196
 - trunc ... 192
 - two commands:
 - \c_thirty_two ... 87, 6129
 - \c_two_hundred_fifty_five .. 87, 6130
 - \c_two_hundred_fifty_six ... 87, 6130
- ## U
- \u ... xxi, 855, 25318
 - \uccode .. 167, 182, 195, 197, 199, 201, 574
 - \Uchar ... 1022
 - \Ucharcat ... 1023
 - \uchyph ... 575
 - \ucs ... 1233
 - \Udelcode ... 1024
 - \Udelcodenum ... 1025
 - \Udelimiter ... 1026

<code>\Udelimiterover</code>	1027	<code>\Umathopclosespacing</code>	1090
<code>\Udelimiterunder</code>	1028	<code>\Umathopenbinspacing</code>	1091
<code>\Uhexensible</code>	1029	<code>\Umathopenclosespacing</code>	1092
<code>\Umathaccent</code>	1030	<code>\Umathopeninnerspacing</code>	1093
<code>\Umathaxis</code>	1031	<code>\Umathopenopenspacing</code>	1094
<code>\Umathbinbinspacing</code>	1032	<code>\Umathopenopspacing</code>	1095
<code>\Umathbinclosespacing</code>	1033	<code>\Umathopenordspacing</code>	1096
<code>\Umathbininnerspacing</code>	1034	<code>\Umathopenpunctspacing</code>	1097
<code>\Umathbinopenspacing</code>	1035	<code>\Umathopenrelspacing</code>	1098
<code>\Umathbinopspacing</code>	1036	<code>\Umathoperatorsize</code>	1099
<code>\Umathbinordspacing</code>	1037	<code>\Umathopinnerspacing</code>	1100
<code>\Umathbinpunctspacing</code>	1038	<code>\Umathopopenspacing</code>	1101
<code>\Umathbinrelspacing</code>	1039	<code>\Umathopopspacing</code>	1102
<code>\Umathchar</code>	1040	<code>\Umathopordspacing</code>	1103
<code>\Umathcharclass</code>	1041	<code>\Umathoppunctspacing</code>	1104
<code>\Umathchardef</code>	1042	<code>\Umathoprelspacing</code>	1105
<code>\Umathcharfam</code>	1043	<code>\Umathordbinspacing</code>	1106
<code>\Umathcharnum</code>	1044	<code>\Umathordclosespacing</code>	1107
<code>\Umathcharnumdef</code>	1045	<code>\Umathordinnerspacing</code>	1108
<code>\Umathcharslot</code>	1046	<code>\Umathordopenspacing</code>	1109
<code>\Umathclosebinspacing</code>	1047	<code>\Umathordopspacing</code>	1110
<code>\Umathcloseclosespacing</code>	1048	<code>\Umathordordspacing</code>	1111
<code>\Umathcloseinnerspacing</code>	1050	<code>\Umathordpunctspacing</code>	1112
<code>\Umathcloseopenspacing</code>	1052	<code>\Umathordrelspacing</code>	1113
<code>\Umathcloseopspacing</code>	1053	<code>\Umathoverbarkern</code>	1114
<code>\Umathcloseordspacing</code>	1054	<code>\Umathoverbarrule</code>	1115
<code>\Umathclosepunctspacing</code>	1055	<code>\Umathoverbarvgap</code>	1116
<code>\Umathcloserelspacing</code>	1057	<code>\Umathoverdelimitervgap</code>	1117
<code>\Umathcode</code>	158, 1058	<code>\Umathpunctbinspacing</code>	1121
<code>\Umathcodenum</code>	1059	<code>\Umathpunctclosespacing</code>	1122
<code>\Umathconnectoroverlapmin</code>	1060	<code>\Umathpunctinnerspacing</code>	1124
<code>\Umathfractiondelsize</code>	1062	<code>\Umathpunctopenspacing</code>	1126
<code>\Umathfractiondenomdown</code>	1063	<code>\Umathpunctopspacing</code>	1127
<code>\Umathfractiondenomvgap</code>	1065	<code>\Umathpunctordspacing</code>	1128
<code>\Umathfractionnumup</code>	1067	<code>\Umathpunctpunctspacing</code>	1129
<code>\Umathfractionnumvgap</code>	1068	<code>\Umathpunctrelspacing</code>	1130
<code>\Umathfractionrule</code>	1069	<code>\Umathquad</code>	1131
<code>\Umathinnerbinspacing</code>	1070	<code>\Umathradicaldegreeafter</code>	1132
<code>\Umathinnerclosespacing</code>	1071	<code>\Umathradicaldegreebefore</code>	1134
<code>\Umathinnerinnerspacing</code>	1073	<code>\Umathradicaldegreeraise</code>	1136
<code>\Umathinneropenspacing</code>	1075	<code>\Umathradicalkern</code>	1138
<code>\Umathinneropspacing</code>	1076	<code>\Umathradicalrule</code>	1139
<code>\Umathinnerordspacing</code>	1077	<code>\Umathradicalvgap</code>	1140
<code>\Umathinnerpunctspacing</code>	1078	<code>\Umathrelbinspacing</code>	1141
<code>\Umathinnerrelspacing</code>	1080	<code>\Umathrelclosespacing</code>	1142
<code>\Umathlimitabovebgap</code>	1081	<code>\Umathrelinnerspacing</code>	1143
<code>\Umathlimitabovekern</code>	1082	<code>\Umathreloppspacing</code>	1144
<code>\Umathlimitabovevgap</code>	1083	<code>\Umathreloppspacing</code>	1145
<code>\Umathlimitbelowbgap</code>	1084	<code>\Umathrelordspacing</code>	1146
<code>\Umathlimitbelowkern</code>	1085	<code>\Umathrelpunctspacing</code>	1147
<code>\Umathlimitbelowvgap</code>	1086	<code>\Umathrelrelspacing</code>	1148
<code>\Umathnolimitsubfactor</code>	1087	<code>\Umathskewedfractionhgap</code>	1149
<code>\Umathnolimitsupfactor</code>	1088	<code>\Umathskewedfractionvgap</code>	1151
<code>\Umathopbinspacing</code>	1089		

<code>\Umathspaceafterscript</code>	1153	8440, 8455, 8464, 8472, 8480, 8486,
<code>\Umathstackdenomdown</code>	1154	8493, 8537, 8547, 8555, 9421, 9499,
<code>\Umathstacknumup</code>	1155	10147, 10877, 10884, 11090, 19107,
<code>\Umathstackvgap</code>	1156	20888, 21033, 23271, 23955, 24030,
<code>\Umathsubshiftdown</code>	1157	24522, 24619, 24628, 24652, 24670
<code>\Umathsubshiftdrop</code>	1158	<code>\use:n</code> 17, 18, 34, 125, 246, 296, 351,
<code>\Umathsubsupshiftdown</code>	1159	461, 517, 518, 605, 627, 786, 796,
<code>\Umathsubsupvgap</code>	1160	875, 914, 1490, 1498, 1572, 1856,
<code>\Umathsubtopmax</code>	1161	1873, 1899, 1945, 1965, 2224, 2301,
<code>\Umathsupbottommin</code>	1162	2761, 2810, 2984, 3234, 3307, 3309,
<code>\Umathsupshiftdrop</code>	1163	3592, 3853, 4035, 4122, 4130, 4220,
<code>\Umathsupshiftdown</code>	1164	4241, 4255, 6531, 6538, 7000, 7372,
<code>\Umathsupsubbottommax</code>	1165	7430, 7512, 7549, 7567, 7613, 8393,
<code>\Umathunderbarkern</code>	1166	8410, 8674, 8691, 9041, 9198, 9248,
<code>\Umathunderbarrule</code>	1167	9341, 9742, 10343, 12013, 12021,
<code>\Umathunderbarvgap</code>	1168	12030, 12047, 12055, 12083, 12540,
<code>\Umathunderdelimitervbgap</code>	1169	14105, 18456, 18655, 19068, 19071,
<code>\Umathunderdelimitervgap</code>	1171	19197, 19729, 19963, 20021, 20100,
undefine commands:		20509, 20561, 20601, 20687, 21419,
<code>.undefine:</code>	170, 11217	21436, 21884, 23526, 23527, 23529,
<code>\underline</code>	576	24050, 24103, 25715, 25801, 25821,
<code>\unexpanded</code>	672	26067, 26496, 26595, 26703, 26715,
<code>\unhbox</code>	577	26727, 27027, 27067, 27078, 27089
<code>\unhcopy</code>	578	<code>\use:nn</code>
<code>\uniformdeviate</code>	1021	17,
<code>\unkern</code>	579	1498, 2479, 3267, 6984, 7838, 10143,
<code>\unless</code>	673	12571, 12580, 12584, 16010, 18424,
<code>\Unosubscript</code>	1173	24063, 24065, 24070, 24072, 24384
<code>\Unosuperscript</code>	1174	<code>\use:nnn</code>
<code>\unpenalty</code>	580	17, 1498, 2278
<code>\unskip</code>	581	<code>\use:nnnn</code>
<code>\unvbox</code>	582	17, 1498
<code>\unvcopy</code>	583	<code>\use_i:nn</code>
<code>\Uoverdelimiter</code>	1175	17, 295,
<code>\uppercase</code>	584	306, 490, 747, 750, 762, 765, 766,
uptex commands:		779, 973, 993, 1445, 1502, 1652,
<code>\uptex_disablecjktoken:D</code>		1745, 1843, 1928, 2065, 2093, 2257,
.....	1227, 5392, 5395, 6550	2967, 3779, 4815, 4817, 5151, 6533,
<code>\uptex_enablecjktoken:D</code>	1228	7968, 9876, 11709, 12540, 13933,
<code>\uptex_forcecjktoken:D</code>	1229	14269, 14562, 15050, 15317, 15834,
<code>\uptex_kchar:D</code>	1230	16005, 16244, 16254, 16258, 16686,
<code>\uptex_kchardef:D</code>	1231, 5396	16891, 17397, 17847, 17890, 17900,
<code>\uptex_kuten:D</code>	1232	17910, 18240, 19028, 19039, 19048,
<code>\uptex_ucs:D</code>	1233	19051, 19060, 24096, 24784, 24891
<code>\uptex_uptexversion:D</code>	1234	<code>\use_i:nnn</code> 17, 779, 1504, 3006, 5017,
<code>\uptexversion</code>	1234	6175, 7847, 12508, 14519, 17391, 20937
<code>\Uradical</code>	1176	<code>\use_i:nnnn</code>
<code>\Uroot</code>	1177	17, 288,
use commands:		439, 440, 1504, 6317, 6332, 6337,
<code>\use:N</code>	15, 90, 308, 1489,	6353, 6355, 14103, 14537, 14544, 14737
.....	1640, 1642, 1690, 1717, 1719, 1912,	<code>\use_i_delimit_by_q_nil:nw</code> . 18, 1515
.....	1994, 2115, 2117, 2119, 2121, 4573,	<code>\use_i_delimit_by_q_recursion_-</code>
.....	5510, 5971, 5981, 6086, 6090, 6092,	stop:nw
.....	6094, 6095, 6099, 6321, 6343, 8429,	18,
		1515, 4619, 4635, 6364, 6379,
		19858, 24569, 24736, 24759, 24808
		<code>\use_i_delimit_by_q_stop:nw</code> . 18,
		386, 1515, 1652, 1654, 2995, 4334,
		4343, 4462, 4513, 4516, 7093, 9462,
		13907, 14269, 17376, 17442, 24120

- \use_i_ii:nnn 1624, 9531, 9540, 12008, 12042, 12068, 12076, 14129, 17394, 17410
- 18, 1504, 2505, 4993, 5092, 8139
- \use_ii:nn \use_none:nnnnnn
 - 17, 100, 295, 306, 490, 747,
750, 762, 765, 766, 778, 799, 804,
860, 1447, 1502, 1748, 1845, 1930,
2067, 2095, 2255, 2448, 3279, 3781,
6539, 7969, 9872, 11910, 11933,
13695, 13933, 13934, 14564, 15836,
16250, 16256, 16260, 16688, 16893,
17324, 18242, 19030, 19036, 19041,
19053, 19062, 19851, 20040, 24437,
24625, 24640, 24783, 24786, 24855
 - 18, 1519, 2001, 9532, 25486
- \use_ii:nnn 17, 1504, 7856
- \use_ii:nnnn . 17, 439, 440, 1504, 6332
- \use_iii:nnn 17, 1504, 2453, 7865, 11715
- \use_iii:nnnn 17,
439, 440, 1504, 6332, 6354, 6356, 6357
- \use_iv:nnnn \use_none:nnnnnnnn
 - 17, 439, 440, 1504, 6332, 6352, 13683
 - 18, 1519
- \use_none:n 18, 289, 356, 363,
363, 454, 457, 500, 531, 624, 625,
628, 628, 800, 806, 813, 1519, 1580,
1696, 1715, 1945, 2226, 2282, 3394,
3415, 3688, 3778, 3794, 3856, 3873,
3883, 3884, 3889, 3904, 3907, 4621,
4636, 4719, 5002, 5841, 5847, 6163,
6532, 6537, 6611, 6655, 6752, 6847,
6874, 6913, 7662, 7926, 7928, 7933,
8328, 8513, 8739, 8743, 9472, 9527,
9582, 10701, 10728, 10751, 10763,
11485, 11704, 11853, 11857, 11861,
11865, 13171, 13397, 13404, 13421,
13440, 13463, 13529, 13570, 13695,
13710, 13731, 13732, 13995, 13996,
14538, 14541, 15505, 16422, 17047,
17332, 17381, 17388, 18238, 18287,
18385, 18423, 18657, 18928, 19074,
19726, 24141, 24142, 24314, 24951,
25583, 25592, 25785, 27042, 27044
- \use_none:nn 18, 301,
354, 359, 368, 368, 402, 1519, 1712,
1752, 1889, 1897, 3377, 3512, 3640,
3805, 3822, 4887, 5024, 6581, 6888,
9528, 9571, 11769, 11852, 11856,
11860, 11864, 17042, 20314, 20950
- \use_none:nnn 18,
369, 1519, 1595, 1604, 1613, 2885,
2900, 3844, 9529, 11851, 11855,
11859, 11863, 12508, 18454, 18783
- \use_none:nnnn \use_none:nnnnnnnnnn
 - 18, 1519, 3038, 9530, 10289
 - 18, 1519, 2001, 9532, 25486
- \use_none:nnnnnn \use_none:nnnnnnnnnn
 - 18, 1519, 1682, 1919, 9533, 12010,
12044, 12070, 12078, 12393, 14578
 - 18, 1519
- \use_none_delimit_by_q_nil:w 18, 1512
- \use_none_delimit_by_q_recursion_
stop:w 18, 62, 62, 62, 62,
1512, 1910, 1980, 1985, 1992, 2801,
2808, 3008, 4613, 4628, 19836, 19860
- \use_none_delimit_by_q_stop:w 18,
393, 456, 1011, 1512, 2999, 4050,
4332, 4341, 4500, 5526, 6834, 7079,
7084, 8566, 10151, 17377, 24049, 25643
- \use_none_delimit_by_s_stop:w ...
..... 63, 64, 4711
- \use_x:n 235, 1496, 4095, 23302
- \useboxresource 1014
- \useimageresource 1015
- \Uskewed 1178
- \Uskewedwithdelims 1179
- \Ustack 1180
- \Ustartdisplaymath 1181
- \Ustartmath 1182
- \Ustopdisplaymath 1183
- \Ustopmath 1184
- \Usubscript 1185
- \Usuperscript 1186
- utex commands:
- \utex_binbinspacing:D 1032
- \utex_binclosespacing:D 1033
- \utex_bininnerspacing:D 1034
- \utex_binopenspacing:D 1035
- \utex_binopspacing:D 1036
- \utex_binordspacing:D 1037
- \utex_binpunctspacing:D 1038
- \utex_binrelspacing:D 1039
- \utex_char:D 1022, 1324, 23461
- \utex_charcat:D 1023, 7333
- \utex_closebinspacing:D 1047
- \utex_closeclosespacing:D 1049
- \utex_closeinnerspacing:D 1051
- \utex_closeopenspacing:D 1052
- \utex_closeopspacing:D 1053
- \utex_closeordspacing:D 1054
- \utex_closepunctspacing:D 1056
- \utex_closerelspacing:D 1057
- \utex_connectoroverlapmin:D .. 1061
- \utex_delcode:D 1024, 1355
- \utex_delcodenum:D 1025, 1356

<code>\utex_delimiter:D</code>	1026, 1357	<code>\utex_oppunctspacing:D</code>	1104
<code>\utex_delimiterover:D</code>	1027	<code>\utex_oprelspacing:D</code>	1105
<code>\utex_delimiterunder:D</code>	1028	<code>\utex_ordbinspacing:D</code>	1106
<code>\utex_fractiondelsize:D</code>	1062	<code>\utex_ordclosespacing:D</code>	1107
<code>\utex_fractiondenomdown:D</code>	1064	<code>\utex_ordinnerspacing:D</code>	1108
<code>\utex_fractiondenomvgap:D</code>	1066	<code>\utex_ordopenspacing:D</code>	1109
<code>\utex_fractionnumup:D</code>	1067	<code>\utex_ordopspacing:D</code>	1110
<code>\utex_fractionnumvgap:D</code>	1068	<code>\utex_ordordspacing:D</code>	1111
<code>\utex_fractionrule:D</code>	1069	<code>\utex_ordpunctspacing:D</code>	1112
<code>\utex_hextensible:D</code>	1029	<code>\utex_ordrelspacing:D</code>	1113
<code>\utex_innerbinspacing:D</code>	1070	<code>\utex_overbarkern:D</code>	1114
<code>\utex_innerclosespacing:D</code>	1072	<code>\utex_overbarrule:D</code>	1115
<code>\utex_innerinnerspacing:D</code>	1074	<code>\utex_overbarvgap:D</code>	1116
<code>\utex_inneropenspacing:D</code>	1075	<code>\utex_overdelimiter:D</code>	1175
<code>\utex_inneropspacing:D</code>	1076	<code>\utex_overdelimiterbgap:D</code>	1118
<code>\utex_innerordspacing:D</code>	1077	<code>\utex_overdelimitervgap:D</code>	1120
<code>\utex_innerpunctspacing:D</code>	1079	<code>\utex_punctbinspacing:D</code>	1121
<code>\utex_innerrelspacing:D</code>	1080	<code>\utex_punctclosespacing:D</code>	1123
<code>\utex_limitabovebgap:D</code>	1081	<code>\utex_punctinnerspacing:D</code>	1125
<code>\utex_limitabovekern:D</code>	1082	<code>\utex_punctopenspacing:D</code>	1126
<code>\utex_limitabovevgap:D</code>	1083	<code>\utex_punctopspacing:D</code>	1127
<code>\utex_limitbelowbgap:D</code>	1084	<code>\utex_punctordspacing:D</code>	1128
<code>\utex_limitbelowkern:D</code>	1085	<code>\utex_punctpunctspacing:D</code>	1129
<code>\utex_limitbelowvgap:D</code>	1086	<code>\utex_punctrelspacing:D</code>	1130
<code>\utex_mathaccent:D</code>	1030, 1358	<code>\utex_quad:D</code>	1131
<code>\utex_mathaxis:D</code>	1031	<code>\utex_radical:D</code>	1176
<code>\utex_mathchar:D</code>	1040, 1359	<code>\utex_radicaldegreeafter:D</code>	1133
<code>\utex_mathcharclass:D</code>	1041	<code>\utex_radicaldegreebefore:D</code>	1135
<code>\utex_mathchardef:D</code>	1042, 1360	<code>\utex_radicaldegreeraise:D</code>	1137
<code>\utex_mathcharfam:D</code>	1043	<code>\utex_radicalkern:D</code>	1138
<code>\utex_mathcharnum:D</code>	1044, 1361	<code>\utex_radicalrule:D</code>	1139
<code>\utex_mathcharnumdef:D</code>	1045, 1362	<code>\utex_radicalvgap:D</code>	1140
<code>\utex_mathcharslot:D</code>	1046	<code>\utex_relbinspacing:D</code>	1141
<code>\utex_mathcode:D</code>	1058, 1363	<code>\utex_relclosespacing:D</code>	1142
<code>\utex_mathcodenum:D</code>	1059, 1364	<code>\utex_relinnerspacing:D</code>	1143
<code>\utex_nolimitsubfactor:D</code>	1087	<code>\utex_reloppspacing:D</code>	1144
<code>\utex_nolimitsupfactor:D</code>	1088	<code>\utex_reloppspacing:D</code>	1145
<code>\utex_nosubscript:D</code>	1173	<code>\utex_relordspacing:D</code>	1146
<code>\utex_nosuperscript:D</code>	1174	<code>\utex_relpunctspacing:D</code>	1147
<code>\utex_opbinspacing:D</code>	1089	<code>\utex_relrelspacing:D</code>	1148
<code>\utex_opclosespacing:D</code>	1090	<code>\utex_root:D</code>	1177
<code>\utex_openbinspacing:D</code>	1091	<code>\utex_skewed:D</code>	1178
<code>\utex_openclosespacing:D</code>	1092	<code>\utex_skewedfractionhgap:D</code>	1150
<code>\utex_openinnerspacing:D</code>	1093	<code>\utex_skewedfractionvgap:D</code>	1152
<code>\utex_openopenspacing:D</code>	1094	<code>\utex_skewedwithdelims:D</code>	1179
<code>\utex_openopspacing:D</code>	1095	<code>\utex_spaceafterscript:D</code>	1153
<code>\utex_openordspacing:D</code>	1096	<code>\utex_stack:D</code>	1180
<code>\utex_openpunctspacing:D</code>	1097	<code>\utex_stackdenomdown:D</code>	1154
<code>\utex_openrelspacing:D</code>	1098	<code>\utex_stacknumup:D</code>	1155
<code>\utex_operatorsize:D</code>	1099	<code>\utex_stackvgap:D</code>	1156
<code>\utex_opinnerspacing:D</code>	1100	<code>\utex_startdisplaymath:D</code>	1181
<code>\utex_opopenspacing:D</code>	1101	<code>\utex_startmath:D</code>	1182
<code>\utex_opopspacing:D</code>	1102	<code>\utex_stopdisplaymath:D</code>	1183
<code>\utex_opordspacing:D</code>	1103	<code>\utex_stopmath:D</code>	1184

[illegible]

\xetex_isexclusivefeature:D	848	\XeTeXglyphbounds	836
\xetex_lastfontchar:D	849	\XeTeXglyphindex	837
\xetex_linebreaklocale:D	851	\XeTeXglyphname	838
\xetex_linebreakpenalty:D	852	\XeTeXinputencoding	839
\xetex_linebreakskip:D	850	\XeTeXinputnormalization	840
\xetex_OTcountfeatures:D	853	\XeTeXinterchartokenstate	842
\xetex_OTcountlanguages:D	854	\XeTeXinterchartoks	844
\xetex_OTcountscripts:D	855	\XeTeXisdefaultselector	845
\xetex_OTfeaturetag:D	856	\XeTeXisexclusivevefeature	847
\xetex_OTlanguagetag:D	857	\XeTeXlastfontchar	849
\xetex_OTscripttag:D	858	\XeTeXlinebreaklocale	851
\xetex_pdffile:D	859, 26476, 26518	\XeTeXlinebreakpenalty	852
\xetex_pdffpagecount:D	860	\XeTeXlinebreakskip	850
\xetex_picfile:D	861, 26469	\XeTeXmathaccent	1358
\xetex_selectorname:D	862	\XeTeXmathchar	1359
\xetex_suppressfontnotfounderror:D		\XeTeXmathchardef	1360
	814, 1325	\XeTeXmathcharnum	1361
\xetex_tracingfonts:D	863	\XeTeXmathcharnumdef	1362
\xetex_upwardsmode:D	864	\XeTeXmathcode	1363
\xetex_useglypmetrics:D	865	\XeTeXmathcodenum	1364
\xetex_variation:D	866	\XeTeXOTcountfeatures	853
\xetex_variationdefault:D	867	\XeTeXOTcountlanguages	854
\xetex_variationmax:D	868	\XeTeXOTcountscripts	855
\xetex_variationmin:D	869	\XeTeXOTfeaturetag	856
\xetex_variationname:D	870	\XeTeXOTlanguagetag	857
\xetex_XeTeXrevision:D	871	\XeTeXOTscripttag	858
\xetex_XeTeXversion:D		\XeTeXpdffile	859
	872, 5394, 6140, 6555	\XeTeXpdffpagecount	860
\XeTeXcharclass	815	\XeTeXpicfile	861
\XeTeXcharglyph	816	\XeTeXrevision	871
\XeTeXcountfeatures	817	\XeTeXselectorname	862
\XeTeXcountglyphs	818	\XeTeXtracingfonts	863
\XeTeXcountselectors	819	\XeTeXupwardsmode	864
\XeTeXcountvariations	820	\XeTeXuseglypmetrics	865
\XeTeXdashbreakstate	822	\XeTeXvariation	866
\XeTeXdefaultencoding	821	\XeTeXvariationdefault	867
\XeTeXdelcode	1354, 1355	\XeTeXvariationmax	868
\XeTeXdelcodenum	1356	\XeTeXvariationmin	869
\XeTeXdelimiter	1357	\XeTeXvariationname	870
\XeTeXfeaturecode	823	\XeTeXversion	872
\XeTeXfeaturename	824	\xkanjiskip	1223
\XeTeXfindfeaturebyname	825	\xleaders	605
\XeTeXfindselectorbyname	827	\xspaceskip	606
\XeTeXfindvariationbyname	829	\xspcode	1224
\XeTeXfirstfontchar	831		
\XeTeXfonttype	832		
\XeTeXgenerateactualtext	833		
\XeTeXglyph	835		
		Y	
		\ybaselineshift	1225
		\year	607
		\yoko	1226