

The L^AT_EX3 Interfaces

The L^AT_EX3 Project*

Released 2017/11/14

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
1.1	Internal functions and variables	7
III	The <code>l3names</code> package: Namespace for primitives	8
1	Setting up the <code>LaTeX3</code> programming language	8
IV	The <code>l3basics</code> package: Basic definitions	9
1	No operation functions	9
2	Grouping material	9
3	Control sequences and functions	10
3.1	Defining functions	10
3.2	Defining new functions using parameter text	11
3.3	Defining new functions using the signature	12
3.4	Copying control sequences	15
3.5	Deleting control sequences	15
3.6	Showing control sequences	15
3.7	Converting to and from control sequences	16
4	Using or removing tokens and arguments	17
4.1	Selecting tokens from delimited arguments	19
5	Predicates and conditionals	19
5.1	Tests on control sequences	20
5.2	Primitive conditionals	21
6	Internal kernel functions	22
V	The <code>l3expan</code> package: Argument expansion	25

1	Defining new variants	25
2	Methods for defining variants	26
3	Introducing the variants	26
4	Manipulating the first argument	28
5	Manipulating two arguments	29
6	Manipulating three arguments	29
7	Unbraced expansion	30
8	Preventing expansion	31
9	Controlled expansion	32
10	Internal functions and variables	33
VI	The <code>l3tl</code> package: Token lists	35
1	Creating and initialising token list variables	35
2	Adding data to token list variables	36
3	Modifying token list variables	37
4	Reassigning token list category codes	37
5	Token list conditionals	38
6	Mapping to token lists	40
7	Using token lists	42
8	Working with the content of token lists	43
9	The first token from a token list	44
10	Using a single item	47
11	Viewing token lists	47
12	Constant token lists	47
13	Scratch token lists	48
14	Internal functions	48
VII	The <code>l3str</code> package: Strings	49

1	Building strings	49
2	Adding data to string variables	50
	2.1 Modifying string variables	51
	2.2 String conditionals	51
3	Mapping to strings	53
4	Working with the content of strings	55
5	String manipulation	58
6	Viewing strings	59
7	Constant token lists	60
8	Scratch strings	60
	8.1 Internal string functions	60
VIII	The <code>l3seq</code> package: Sequences and stacks	62
1	Creating and initialising sequences	62
2	Appending data to sequences	63
3	Recovering items from sequences	63
4	Recovering values from sequences with branching	64
5	Modifying sequences	65
6	Sequence conditionals	66
7	Mapping to sequences	66
8	Using the content of sequences directly	68
9	Sequences as stacks	69
10	Sequences as sets	70
11	Constant and scratch sequences	71
12	Viewing sequences	72
13	Internal sequence functions	72
IX	The <code>l3int</code> package: Integers	73
1	Integer expressions	73

2	Creating and initialising integers	74
3	Setting and incrementing integers	75
4	Using integers	75
5	Integer expression conditionals	76
6	Integer expression loops	77
7	Integer step functions	79
8	Formatting integers	79
9	Converting from other formats to integers	81
10	Viewing integers	82
11	Constant integers	83
12	Scratch integers	83
13	Primitive conditionals	84
14	Internal functions	84
X	The <code>l3intarray</code> package: low-level arrays of small integers	86
1	<code>l3intarray</code> documentation	86
1.1	Internal functions	86
XI	The <code>l3flag</code> package: expandable flags	87
1	Setting up flags	87
2	Expandable flag commands	88
XII	The <code>l3quark</code> package: Quarks	89
1	Introduction to quarks and scan marks	89
1.1	Quarks	89
2	Defining quarks	89
3	Quark tests	90
4	Recursion	90
5	An example of recursion with quarks	91

6	Internal quark functions	92
7	Scan marks	92
XIII The <code>l3prg</code> package: Control structures		94
1	Defining a set of conditional functions	94
2	The boolean data type	96
3	Boolean expressions	98
4	Logical loops	100
5	Producing multiple copies	101
6	Detecting TeX's mode	101
7	Primitive conditionals	101
8	Internal programming functions	102
XIV The <code>l3clist</code> package: Comma separated lists		103
1	Creating and initialising comma lists	103
2	Adding data to comma lists	104
3	Modifying comma lists	105
4	Comma list conditionals	106
5	Mapping to comma lists	106
6	Using the content of comma lists directly	108
7	Comma lists as stacks	109
8	Using a single item	110
9	Viewing comma lists	110
10	Constant and scratch comma lists	111
XV The <code>l3token</code> package: Token manipulation		112
1	Creating character tokens	112
2	Manipulating and interrogating character tokens	114
3	Generic tokens	117

4	Converting tokens	117
5	Token conditionals	118
6	Peeking ahead at the next token	121
7	Decomposing a macro definition	123
8	Description of all possible tokens	124
9	Internal functions	126
XVI The <code>l3prop</code> package: Property lists		127
1	Creating and initialising property lists	127
2	Adding entries to property lists	128
3	Recovering values from property lists	128
4	Modifying property lists	129
5	Property list conditionals	129
6	Recovering values from property lists with branching	130
7	Mapping to property lists	130
8	Viewing property lists	131
9	Scratch property lists	132
10	Constants	132
11	Internal property list functions	132
XVII The <code>l3msg</code> package: Messages		133
1	Creating new messages	133
2	Contextual information for messages	134
3	Issuing messages	135
4	Redirecting messages	137
5	Low-level message functions	138
6	Kernel-specific functions	139
7	Expandable errors	141

8	Internal l3msg functions	141
XVIII	The l3file package: File and I/O operations	143
1	File operation functions	143
1.1	Input-output stream management	144
1.2	Reading from files	146
2	Writing to files	148
2.1	Wrapping lines in output	150
2.2	Constant input-output streams	151
2.3	Primitive conditionals	151
2.4	Internal file functions and variables	151
2.5	Internal input-output functions	152
XIX	The l3skip package: Dimensions and skips	153
1	Creating and initialising dim variables	153
2	Setting dim variables	154
3	Utilities for dimension calculations	154
4	Dimension expression conditionals	155
5	Dimension expression loops	157
6	Using dim expressions and variables	158
7	Viewing dim variables	160
8	Constant dimensions	160
9	Scratch dimensions	160
10	Creating and initialising skip variables	161
11	Setting skip variables	161
12	Skip expression conditionals	162
13	Using skip expressions and variables	162
14	Viewing skip variables	162
15	Constant skips	163
16	Scratch skips	163
17	Inserting skips into the output	163

18	Creating and initialising muskip variables	164
19	Setting muskip variables	164
20	Using muskip expressions and variables	165
21	Viewing muskip variables	165
22	Constant muskips	166
23	Scratch muskips	166
24	Primitive conditional	166
25	Internal functions	167
XX	The l3keys package: Key–value interfaces	168
1	Creating keys	169
2	Sub-dividing keys	173
3	Choice and multiple choice keys	173
4	Setting keys	176
5	Handling of unknown keys	176
6	Selective key setting	177
7	Utility functions for keys	178
8	Low-level interface for parsing key–val lists	179
XXI	The l3fp package: floating points	181
1	Creating and initialising floating point variables	182
2	Setting floating point variables	182
3	Using floating point numbers	183
4	Floating point conditionals	184
5	Floating point expression loops	186
6	Some useful constants, and scratch variables	187
7	Floating point exceptions	188
8	Viewing floating points	189

9	Floating point expressions	189
9.1	Input of floating point numbers	189
9.2	Precedence of operators	190
9.3	Operations	191
10	Disclaimer and roadmap	197
XXII	The <code>l3sort</code> package: Sorting functions	200
1	Controlling sorting	200
XXIII	The <code>l3tl-build</code> package: building token lists	201
1	<code>l3tl-build</code> documentation	201
1.1	Internal functions	201
XXIV	The <code>l3tl-analysis</code> package: analysing token lists	202
1	<code>l3tl-analysis</code> documentation	202
XXV	The <code>l3regex</code> package: regular expressions in \TeX	203
1	Regular expressions	203
1.1	Syntax of regular expressions	203
1.2	Syntax of the replacement text	208
1.3	Pre-compiling regular expressions	210
1.4	Matching	210
1.5	Submatch extraction	211
1.6	Replacement	212
1.7	Bugs, misfeatures, future work, and other possibilities	212
XXVI	The <code>l3box</code> package: Boxes	216
1	Creating and initialising boxes	216
2	Using boxes	217
3	Measuring and setting box dimensions	218
4	Box conditionals	218
5	The last box inserted	219
6	Constant boxes	219
7	Scratch boxes	219

8	Viewing box contents	219
9	Boxes and color	220
10	Horizontal mode boxes	220
11	Vertical mode boxes	221
12	Affine transformations	223
13	Primitive box conditionals	225
XXVII The l3coffins package: Coffin code layer		227
1	Creating and initialising coffins	227
2	Setting coffin content and poles	227
3	Joining and using coffins	228
4	Measuring coffins	229
5	Coffin diagnostics	229
	5.1 Constants and variables	230
XXVIII The l3color package: Color support		231
1	Color in boxes	231
	1.1 Internal functions	232
XXIX The l3sys package: System/runtime functions		233
1	The name of the job	233
2	Date and time	233
3	Engine	233
4	Output format	234
XXX The l3deprecation package: Deprecation errors		235
1	l3deprecation documentation	235
XXXI The l3candidates package: Experimental additions to l3kernel		236
1	Important notice	236

2	Additions to l3basics	237
3	Additions to l3box	237
	3.1 Viewing part of a box	237
4	Additions to l3clist	238
5	Additions to l3coffins	238
6	Additions to l3file	239
7	Additions to l3int	240
8	Additions to l3msg	240
9	Additions to l3prop	240
10	Additions to l3seq	241
11	Additions to l3skip	242
12	Additions to l3sys	242
13	Additions to l3tl	243
14	Additions to l3token	249
XXXII	The l3luatex package: LuaTeX-specific functions	250
1	Breaking out to Lua	250
	1.1 T _E X code interfaces	250
	1.2 Lua interfaces	251
XXXIII	The l3drivers package: Drivers	252
1	Box clipping	252
2	Box rotation and scaling	252
3	Color support	253
4	Drawing	253
	4.1 Path construction	254
	4.2 Stroking and filling	254
	4.3 Stroke options	255
	4.4 Color	256
	4.5 Inserting T _E X material	257
	4.6 Coordinate system transformations	257

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 `LATEX3` programming language is found in [expl3.pdf](#).

1 Naming functions and variables

`LATEX3` 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 `TEX` 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 *c_{name}*, and indicates that the argument will be turned into a `cname` 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 `TEX` 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 `cname` 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 `TEX` `\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).
- t1** 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
```

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:

`\seq_new:N`
`\seq_new:c`

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

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:

`\cs_to_str:N` ☆

`\cs_to_str:N` $\langle cs \rangle$

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:

`\seq_map_function:NN` ☆

`\seq_map_function:NN` $\langle seq \rangle$ $\langle function \rangle$

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:

`\sys_if_engine_xetex:TF` ☆

`\sys_if_engine_xetex:TF` $\langle true\ code \rangle$ $\langle false\ code \rangle$

The underlining and italic of **TF** indicates that `\sys_if_engine_xetex:T`, `\sys_if_engine_xetex:F` and `\sys_if_engine_xetex:TF` are all available. 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:

`\l_tmpa_tl`

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:

`\token_to_str:N` ☆

`\token_to_str:N` $\langle token \rangle$

The normal description text.

\TeX hackers note: Detail for the experienced \TeX or $\text{\LaTeX} 2_{\epsilon}$ programmer. In this case, it would point out that this function is the \TeX 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 `LATEX3` 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 *⟨true code⟩* or the *⟨false code⟩* 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 `TEX` concepts not supported by `LATEX3`

The `TEX` concept of an “`\outer`” macro is *not supported* at all by `LATEX3`. As such, the functions provided here may break when used on top of `LATEX 2ε` 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` `\ExplSyntaxOn <code> \ExplSyntaxOff`

`\ExplSyntaxOff`

Updated: 2011-08-13

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}
```

1.1 Internal functions and variables

\l_kernel_expl_bool

A boolean which records the current code syntax status: `true` if currently inside a code environment. This variable should only be set by `\ExplSyntaxOn/\ExplSyntaxOff`.

Part III

The `l3names` package Namespace for primitives

1 Setting up the \LaTeX 3 programming language

This module is at the core of the \LaTeX 3 programming language. It performs the following tasks:

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

This module is entirely dedicated to primitives, which should not be used directly within \LaTeX 3 code (outside of “kernel-level” code). As such, the primitives are not documented here: *The \TeX book*, *\TeX by Topic* and the manuals for $\text{pdf}\TeX$, $\text{X}\TeX$ and $\text{Lua}\TeX$ should be consulted for details of the primitives. These are named based on the engine which first introduced them:

`\tex_...` Introduced by \TeX itself;
`\etex_...` Introduced by the ε - \TeX extensions;
`\pdftex_...` Introduced by $\text{pdf}\TeX$;
`\xetex_...` Introduced by $\text{X}\TeX$;
`\luatex_...` Introduced by $\text{Lua}\TeX$;
`\utex_...` Introduced by $\text{X}\TeX$ and $\text{Lua}\TeX$;
`\ptex_...` Introduced by $\text{p}\TeX$;
`\uptex_...` Introduced by $\text{up}\TeX$.

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` $\langle token \rangle$

Adds $\langle token \rangle$ to the list of $\langle tokens \rangle$ to be inserted when the current group level ends. The list of $\langle tokens \rangle$ 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 $\langle token \rangle$ 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 \mathbf{x} expansion. In contrast, “protected” functions are not expanded within \mathbf{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 $\backslash\text{cs_new}\dots$ 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 ($\#1$, $\#2$, \dots).

new Create a new function with the **new** scope, such as $\backslash\text{cs_new:Npn}$. 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 $\backslash\text{cs_set:Npn}$. 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 $\backslash\text{cs_gset:Npn}$. 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 $\backslash\text{cs_set_nopar:Npn}$. The parameter may not contain $\backslash\text{par}$ tokens.

protected Create a new function with the **protected** restriction, such as $\backslash\text{cs_set_protected:Npn}$. The parameter may contain $\backslash\text{par}$ tokens but the function will not expand within an \mathbf{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 $\backslash\text{prg_new_conditional}$: functions described in Section 1).

p and w These are special cases.

The $\backslash\text{cs_new}$: 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 $\backslash\text{cs_generate_variant:Nn}$ 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 <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the
<code>\cs_new:Npx</code>	<i><parameters></i> (#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 <i><function></i> 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 <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the
<code>\cs_new_nopar:Npx</code>	<i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the
<code>\cs_new_nopar:cpx</code>	<i><function></i> is used the <i><parameters></i> absorbed cannot contain <code>\par</code> tokens. The definition is global and an error results if the <i><function></i> 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 <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the
<code>\cs_new_protected:Npx</code>	<i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_new_protected:cpx</code>	<i><function></i> will not expand within an x-type argument. The definition is global and an error results if the <i><function></i> 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 <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the
<code>\cs_set:Npx</code>	<i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. The
<code>\cs_set:cpx</code>	assignment of a meaning to the <i><function></i> 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 <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the
<code>\cs_set_nopar:Npx</code>	<i><parameters></i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the
<code>\cs_set_nopar:cpx</code>	<i><function></i> is used the <i><parameters></i> absorbed cannot contain <code>\par</code> tokens. The assignment of a meaning to the <i><function></i> 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 <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the
<code>\cs_set_protected:Npx</code>	<i><parameters></i> (#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 <i><function></i> is restricted to the current \TeX group level. The <i><function></i> will not expand within an x-type argument.

```

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

```

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 $\backslash\text{par}$ tokens. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current $\text{T}_{\text{E}}\text{X}$ group level. The $\langle function \rangle$ will not expand within an x -type argument.

```

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

```

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 $\text{T}_{\text{E}}\text{X}$ group level: the assignment is global.

```

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

```

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 $\backslash\text{par}$ tokens. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current $\text{T}_{\text{E}}\text{X}$ group level: the assignment is global.

```

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

```

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 $\text{T}_{\text{E}}\text{X}$ group level: the assignment is global. The $\langle function \rangle$ will not expand within an x -type argument.

```

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

```

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 $\backslash\text{par}$ tokens. The assignment of a meaning to the $\langle function \rangle$ is *not* restricted to the current $\text{T}_{\text{E}}\text{X}$ 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

```

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

```

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.

$\backslash\text{cs_new_nopar:Nn}$ $\backslash\text{cs_new_nopar:(cn Nx cx)}$	$\backslash\text{cs_new_nopar:Nn}$ $\langle function \rangle$ $\{\langle code \rangle\}$ 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 $\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain $\backslash\text{par}$ tokens. The definition is global and an error results if the $\langle function \rangle$ is already defined.
--	--

$\backslash\text{cs_new_protected:Nn}$ $\backslash\text{cs_new_protected:(cn Nx cx)}$	$\backslash\text{cs_new_protected:Nn}$ $\langle function \rangle$ $\{\langle code \rangle\}$ 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 $\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) 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.
--	--

$\backslash\text{cs_new_protected_nopar:Nn}$ $\backslash\text{cs_new_protected_nopar:(cn Nx cx)}$	$\backslash\text{cs_new_protected_nopar:Nn}$ $\langle function \rangle$ $\{\langle code \rangle\}$ 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 $\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain $\backslash\text{par}$ 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.
--	---

$\backslash\text{cs_set:Nn}$ $\backslash\text{cs_set:(cn Nx cx)}$	$\backslash\text{cs_set:Nn}$ $\langle function \rangle$ $\{\langle code \rangle\}$ 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 $\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) will be replaced by those absorbed by the function. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current $\text{T}_{\text{E}}\text{X}$ group level.
--	--

$\backslash\text{cs_set_nopar:Nn}$ $\backslash\text{cs_set_nopar:(cn Nx cx)}$	$\backslash\text{cs_set_nopar:Nn}$ $\langle function \rangle$ $\{\langle code \rangle\}$ 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 $\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) will be replaced by those absorbed by the function. When the $\langle function \rangle$ is used the $\langle parameters \rangle$ absorbed cannot contain $\backslash\text{par}$ tokens. The assignment of a meaning to the $\langle function \rangle$ is restricted to the current $\text{T}_{\text{E}}\text{X}$ group level.
--	---

$\backslash\text{cs_set_protected:Nn}$ $\backslash\text{cs_set_protected:(cn Nx cx)}$	$\backslash\text{cs_set_protected:Nn}$ $\langle function \rangle$ $\{\langle code \rangle\}$ 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 $\langle parameters \rangle$ ($\#1$, $\#2$, <i>etc.</i>) 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 $\text{T}_{\text{E}}\text{X}$ group level.
--	---

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

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<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 assignment of a meaning to the *<function>* is restricted to the current TeX group level.

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

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

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

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<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 assignment of a meaning to the *<function>* is global.

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

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

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

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<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 assignment of a meaning to the *<function>* is global.

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

Updated: 2012-01-14

Uses the *<creator>* function (which should have signature `Npn`, for example `\cs_new:Npn`) to define a *<function>* which takes *<number>* arguments and has *<code>* as replacement text. The *<number>* 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”.

<code>\cs_new_eq:NN</code>	<code>\cs_new_eq:NN <cs₁> <cs₂></code>
<code>\cs_new_eq:(Nc cN cc)</code>	<code>\cs_new_eq:NN <cs₁> <token></code>

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.

<code>\cs_set_eq:NN</code>	<code>\cs_set_eq:NN <cs₁> <cs₂></code>
<code>\cs_set_eq:(Nc cN cc)</code>	<code>\cs_set_eq:NN <cs₁> <token></code>

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 $\text{T}_{\text{E}}\text{X}$ group level.

<code>\cs_gset_eq:NN</code>	<code>\cs_gset_eq:NN <cs₁> <cs₂></code>
<code>\cs_gset_eq:(Nc cN cc)</code>	<code>\cs_gset_eq:NN <cs₁> <token></code>

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 $\text{T}_{\text{E}}\text{X}$ 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.

<code>\cs_undefine:N</code>	<code>\cs_undefine:N <control\ sequence></code>
<code>\cs_undefine:c</code>	Sets $\langle control\ sequence \rangle$ to be globally undefined.

Updated: 2011-09-15

3.6 Showing control sequences

<code>\cs_meaning:N *</code>	<code>\cs_meaning:N <control\ sequence></code>
<code>\cs_meaning:c *</code>	This function expands to the <i>meaning</i> of the $\langle control\ sequence \rangle$ control sequence. For a macro, this includes the $\langle replacement\ text \rangle$.

Updated: 2011-12-22

$\text{T}_{\text{E}}\text{X}$ hackers note: This is $\text{T}_{\text{E}}\text{X}$'s `\meaning` primitive. 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.

TeXhackers note: This is similar to the TeX 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` $\{\langle control\ sequence\ name \rangle\}$

Converts the given $\langle control\ sequence\ name \rangle$ into a single control sequence token. This process requires two expansions. The content for $\langle control\ sequence\ name \rangle$ may be literal material or from other expandable functions. The $\langle control\ sequence\ name \rangle$ 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.

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:c` $\langle control\ sequence \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests whether the $\langle control\ sequence \rangle$ is currently defined (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.

`\cs:w` ★
`\cs_end:` ★

`\cs:w` $\langle control\ sequence\ name \rangle$ `\cs_end:`

Converts the given $\langle control\ sequence\ name \rangle$ into a single control sequence token. This process requires one expansion. The content for $\langle control\ sequence\ name \rangle$ may be literal material or from other expandable functions. The $\langle control\ sequence\ name \rangle$ 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.

TeXhackers note: These are the TeX 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`.

```
\cs_to_str:N * \cs_to_str:N <control sequence>
```

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 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).

```
\use:n * \use:n {<group1>}  
\use:nn * \use:nn {<group1>} {<group2>}  
\use:nnn * \use:nnn {<group1>} {<group2>} {<group3>}  
\use:nnnn * \use:nnnn {<group1>} {<group2>} {<group3>} {<group4>}
```

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.

`\use_i:nn` * `\use_i:nn {<arg1>} {<arg2>}`

`\use_ii:nn` *

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.

`\use_i:nnn` * `\use_i:nnn {<arg1>} {<arg2>} {<arg3>}`

`\use_ii:nnn` *

`\use_iii:nnn` *

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.

`\use_i:nnnn` * `\use_i:nnnn {<arg1>} {<arg2>} {<arg3>} {<arg4>}`

`\use_ii:nnnn` *

`\use_iii:nnnn` *

`\use_iv:nnnn` *

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.

`\use_i_ii:nnn` * `\use_i_ii:nnn {<arg1>} {<arg2>} {<arg3>}`

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.

`\use_none:n` * `\use_none:n {<group1>}`

`\use_none:nn` *

`\use_none:nnn` *

`\use_none:nnnn` *

`\use_none:nnnnn` *

`\use_none:nnnnnn` *

`\use_none:nnnnnnn` *

`\use_none:nnnnnnnn` *

`\use_none:nnnnnnnnn` *

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).

`\use:x` `\use:x {⟨expandable tokens⟩}`

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.

4.1 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⟩</code> <code>\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_recursion_stop:nw {⟨inserted tokens⟩} ⟨balanced</code> <code>text⟩ \q_stop</code> <code>\use_i_delimit_by_q_recursion_stop:nw {⟨inserted tokens⟩}</code> <code>⟨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.

5 Predicates and conditionals

LaTeX3 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 \TeX and $\LaTeX 2_\epsilon$. 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.

5.1 Tests on control sequences

```
\cs_if_eq_p:NN * \cs_if_eq_p:NN <cs1> <cs2>
\cs_if_eq:NNTF * \cs_if_eq:NNTF <cs1> <cs2> {\true code} {\false 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`.

```
\cs_if_exist_p:N * \cs_if_exist_p:N <control sequence>
\cs_if_exist_p:c * \cs_if_exist:NNTF <control sequence> {\true code} {\false code}
\cs_if_exist:NNTF *
\cs_if_exist:cTF * Tests whether the <control sequence> is currently defined (whether as a function or another
control sequence type). Any valid definition of <control sequence> evaluates as true.
```

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

5.2 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 <code><true code></code> , while <code>\if_false:</code> always executes <code><false code></code> .
<code>\reverse_if:N</code>	★	<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>\int</code> and used in case switches.

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

<code>\if_meaning:w</code>	★	<code>\if_meaning:w <arg₁₂</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.`

TeXhackers note: This is TeX's `\ifx`.

<code>\if:w</code>	★	<code>\if:w <token₁₂</code>
<code>\if_charcode:w</code>	★	<code>\if_catcode:w <token₁₂</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>	★	

6 Internal kernel functions

`_chk_if_free_cs:N`
`_chk_if_free_cs:c`

`_chk_if_free_cs:N` $\langle cs \rangle$

This function checks that $\langle cs \rangle$ is free according to the criteria for `\cs_if_free_p:N`, and if not raises a kernel-level error.

`_cs_count_signature:N *`
`_cs_count_signature:c *`

`_cs_count_signature:N` $\langle function \rangle$

Splits the $\langle function \rangle$ into the $\langle name \rangle$ (*i.e.* the part before the colon) and the $\langle signature \rangle$ (*i.e.* after the colon). The $\langle number \rangle$ of tokens in the $\langle signature \rangle$ is then left in the input stream. If there was no $\langle signature \rangle$ then the result is the marker value -1 .

`_cs_split_function:NN *`

`_cs_split_function:NN` $\langle function \rangle$ $\langle processor \rangle$

Splits the $\langle function \rangle$ into the $\langle name \rangle$ (*i.e.* the part before the colon) and the $\langle signature \rangle$ (*i.e.* after the colon). This information is then placed in the input stream after the $\langle processor \rangle$ function in three parts: the $\langle name \rangle$, the $\langle signature \rangle$ and a logic token indicating if a colon was found (to differentiate variables from function names). The $\langle name \rangle$ does not include the escape character, and both the $\langle name \rangle$ and $\langle signature \rangle$ are made up of tokens with category code 12 (other). The $\langle processor \rangle$ should be a function with argument specification `:nnN` (plus any trailing arguments needed).

`_cs_get_function_name:N *` `_cs_get_function_name:N` $\langle function \rangle$

Splits the $\langle function \rangle$ into the $\langle name \rangle$ (*i.e.* the part before the colon) and the $\langle signature \rangle$ (*i.e.* after the colon). The $\langle name \rangle$ 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` $\langle function \rangle$

Splits the $\langle function \rangle$ into the $\langle name \rangle$ (*i.e.* the part before the colon) and the $\langle signature \rangle$ (*i.e.* after the colon). The $\langle signature \rangle$ is then left in the input stream made up of tokens with category code 12 (other).

`_cs_tmp:w`

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).

`_debug:TF`

`_debug:TF` $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Runs the $\langle true code \rangle$ if debugging is enabled, namely only in L^AT_EX 2_ε package mode with one of the options `check-declarations`, `enable-debug`, or `log-functions`. Otherwise runs the $\langle false code \rangle$. The T and F variants are not provided for this low-level conditional.

`_debug_chk_cs_exist:N`
`_debug_chk_cs_exist:c`

`_debug_chk_cs_exist:N` $\langle cs \rangle$

This function is only created if debugging is enabled. It checks that $\langle cs \rangle$ exists according to the criteria for `\cs_if_exist_p:N`, and if not raises a kernel-level error.

`__debug_chk_expr:nNn` `__debug_chk_expr:nNn {<expr>} {<eval>} {<convert>} {<caller>}`

This function is only created if debugging is enabled. By default it is equivalent to `\use_i:nmmn`. When expression checking is enabled, it leaves in the input stream the result of `\tex_the:D {<eval>} {<expr>} \tex_relax:D` 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 `<caller>`. For instance `<eval>` can be `__int_eval:w` and `<caller>` can be `\int_eval:n` or `\int_set:Nn`. The argument `<convert>` is empty except for mu expressions where it is `\etex_mutoglua:D`, used for internal purposes.

`__debug_chk_var_exist:N` `__debug_chk_var_exist:N {<var>}`

This function is only created if debugging is enabled. It checks that `<var>` is defined according to the criteria for `\cs_if_exist_p:N`, and if not raises a kernel-level error.

`__debug_log:x` `__debug_log:x {<message text>}`

If the `log-functions` option is active, this function writes the `<message text>` to the log file using `\iow_log:x`. Otherwise, the `<message text>` is ignored using `\use_none:n`. This function is only created if debugging is enabled.

`__debug_suspend_log:` `__debug_suspend_log: ... __debug_log:x ... __debug_resume_log:`
`__debug_resume_log:`

Any `__debug_log:x` command between `__debug_suspend_log:` and `__debug_resume_log:` is suppressed. These two commands can be nested. These functions are only created if debugging is enabled.

`__debug_patch:nnNpn` `__debug_patch:nnNpn {<before>} {<after>} {<definition>} {<function>} {<parameters>} {<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.

`__debug_patch_conditional:nnNpnn` `__debug_patch_conditional:nnNpnn {<before>} {<definition>} {<conditional>} {<parameters>} {<type>} {<code>}`

Similar to `__debug_patch:nnNpn` 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.

<code>__debug_patch_args:nNNpn</code>	<code>__debug_patch_args:nNNpn {<arguments>}</code>
<code>__debug_patch_conditional_args:nNNpnn</code>	<code><definition> <function> <parameters> {<code>}</code>

Like `__debug_patch:nnNNpn`, 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,

```
\__debug_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 `__debug_patch_conditional_args:nNNpnn` function is for use before `\prg_new_conditional:Npnn` or its `_protected` counterpart.

<code>__kernel_register_show:N</code>	<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.

<code>__kernel_register_log:N</code>	<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`.

Updated: 2015-08-03

<code>__prg_case_end:nw</code> ★	<code>__prg_case_end:nw {<code>} <tokens> \q_mark {<>true code>} \q_mark {<>false code>} \q_stop</code>
-----------------------------------	--

Used to terminate case statements (`\int_case:nnTF`, *etc.*) by removing trailing `<tokens>` and the end marker `\q_stop`, inserting the `<code>` for the successful case (if one is found) and either the `true code` or `false code` for the over all outcome, as appropriate.

Part V

The `l3expan` package

Argument expansion

This module provides generic methods for expanding \TeX 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 \LaTeX 3 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 uncritical 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`, `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`

Updated: 2015-08-06

`\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 where these are not already defined. For each \langle variant \rangle given, a function is created which 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. Only `n` and `N` arguments can be changed to other types. 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.

While `\cs_generate_variant:Nn \foo:N { o }` is currently allowed, one must know that it will break if the result of the expansion is more than one token or if `\foo:N` requires its argument not to be braced.

3 Introducing the variants

The available internal functions for argument expansion come in two flavours, some of them are faster than others. Therefore (when speed is important) it is usually best to follow the following guidelines when defining new functions that are supposed to come with variant forms:

- Arguments that might need expansion should come first in the list of arguments to make processing faster.
- Arguments that should consist of single tokens should come first.
- 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.
- In general, unless in the last position, multi-token arguments **n**, **f**, and **o** need special processing when more than one argument is being expanded. This special processing is not fast. 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**, which are not expanded.

The **V** type returns the value of a register, which can be one of **tl**, **int**, **skip**, **dim**, **toks**, 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 it does not try to 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 *emph*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.

```
\exp_args:No * \exp_args:No <function> {<tokens>} ...
```

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.

```
\exp_args:Nc * \exp_args:Nc <function> {<tokens>}
```

```
\exp_args:cc *
```

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. (An internal error occurs if such a conversion is not possible). 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>`.

```
\exp_args:Nv * \exp_args:Nv <function> <variable>
```

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. (An internal error occurs if such a conversion is not possible). 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.

```
\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 or space is found, 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>* second. 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:NNo` * `\exp_args:NNc` `<token1>` `<token2>` `{<tokens>}`

`\exp_args:NNc` *

`\exp_args:NNv` *

`\exp_args:NNV` *

`\exp_args:NNf` *

`\exp_args:Nco` *

`\exp_args:Ncf` *

`\exp_args:Ncc` *

`\exp_args:NVV` *

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:Nno` * `\exp_args:Noo` `<token>` `{<tokens1>}` `{<tokens2>}`

`\exp_args:NnV` *

`\exp_args:Nnf` *

`\exp_args:Noo` *

`\exp_args:Nof` *

`\exp_args:Noc` *

`\exp_args:Nff` *

`\exp_args:Nfo` *

`\exp_args:Nnc` *

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 special (slower) processing.

Updated: 2012-01-14

`\exp_args:NNx` `\exp_args:NNx <token1>` `<token2>` `{<tokens>}`

`\exp_args:Nnx`

`\exp_args:Ncx`

`\exp_args:Nox`

`\exp_args:Nxo`

`\exp_args:Nxx`

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.

6 Manipulating three arguments

`\exp_args:NNNo` * `\exp_args:NNNo <token1>` `<token2>` `<token3>` `{<tokens>}`

`\exp_args:NNNV` *

`\exp_args:Nccc` *

`\exp_args:NcNc` *

`\exp_args:NcNo` *

`\exp_args:Ncco` *

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, *etc.*

<code>\exp_args:NNoo</code>	★	<code>\exp_args:NNoo</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\{\langle tokens_3 \rangle\}$	$\{\langle tokens \rangle\}$
<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 special (slower) processing.				
<code>\exp_args:Nnno</code>	★					
<code>\exp_args:Nnnc</code>	★					
<code>\exp_args:Nooo</code>	★					

<code>\exp_args:NNNx</code>		<code>\exp_args:NNNx</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$	$\{\langle tokens_1 \rangle\}$	$\{\langle tokens_2 \rangle\}$
<code>\exp_args:NNnx</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>				
<code>\exp_args:NNox</code>						
<code>\exp_args:Nnnx</code>						
<code>\exp_args:Nnox</code>						
<code>\exp_args:Noox</code>						
<code>\exp_args:Ncnx</code>						
<code>\exp_args:Nccx</code>						

New: 2015-08-12

7 Unbraced expansion

<code>\exp_last_unbraced:NV</code>	★	<code>\exp_last_unbraced:Nno</code>	$\langle token \rangle$	$\{\langle tokens_1 \rangle\}$	$\{\langle tokens_2 \rangle\}$
<code>\exp_last_unbraced:(Nf No Nv)</code>	★				
<code>\exp_last_unbraced:Nco</code>	★				
<code>\exp_last_unbraced:(NcV NNV NNo)</code>	★				
<code>\exp_last_unbraced:Nno</code>	★				
<code>\exp_last_unbraced:(Noo Nfo)</code>	★				
<code>\exp_last_unbraced:NNNV</code>	★				
<code>\exp_last_unbraced:NNNo</code>	★				
<code>\exp_last_unbraced:NnNo</code>	★				

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`, and `:Nfo` variants need special (slower) processing.

TeXhackers 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.

<code>\exp_last_unbraced:Nx</code>		<code>\exp_last_unbraced:Nx</code>	$\langle function \rangle$	$\{\langle tokens \rangle\}$
------------------------------------	--	------------------------------------	----------------------------	------------------------------

This functions fully expands the $\langle tokens \rangle$ and leaves the result in the input stream after reinsertion of $\langle function \rangle$. This function is not expandable.

<code>\exp_last_two_unbraced:Noo</code>	★	<code>\exp_last_two_unbraced:Noo</code>	$\langle token \rangle$	$\{\langle tokens_1 \rangle\}$	$\{\langle tokens_2 \rangle\}$
---	---	---	-------------------------	--------------------------------	--------------------------------

This function absorbs three arguments and expand 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, 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.

\TeX hackers note: This is the \TeX `\noexpand` primitive.

`\exp_not:c` ★

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

Expands the $\langle tokens \rangle$ until only unexpandable content remains, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited.

`\exp_not:n` ★

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

Prevents expansion of the $\langle tokens \rangle$ in a context where they would otherwise be expanded, for example an `x`-type argument.

\TeX hackers note: This is the ε - \TeX `\unexpanded` primitive. Hence its argument *must* be surrounded by braces.

`\exp_not:V` ★

`\exp_not:V` $\langle variable \rangle$

Recovers the content of the $\langle variable \rangle$, then prevents expansion of this material in a context where it would otherwise be expanded, for example an `x`-type argument.

`\exp_not:v` ★

`\exp_not:v` $\{ \langle tokens \rangle \}$

Expands the $\langle tokens \rangle$ until only unexpandable content remains, and then converts this into a control sequence (which should be a $\langle variable \rangle$ name). The content of the $\langle variable \rangle$ is recovered, and further expansion is prevented in a context where it would otherwise be expanded, for example an `x`-type argument.

`\exp_not:o` ★

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

Expands the $\langle tokens \rangle$ once, then prevents any further expansion in a context where they would otherwise be expanded, for example an `x`-type argument.

`\exp_not:f` ★ `\exp_not:f {⟨tokens⟩}`

Expands `⟨tokens⟩` fully until the first unexpandable token is found. Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion.

`\exp_stop_f` ★ `\foo_bar:f { ⟨tokens⟩ \exp_stop_f: ⟨more tokens⟩ }`

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 `TEX` 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 `TEX` 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:w ⟨expandable-tokens⟩ \exp_end:`

`\exp_end:` ★

New: 2015-08-23

Expands `⟨expandable-tokens⟩` until reaching `\exp_end:` at which point expansion stops. The full expansion of `⟨expandable-tokens⟩` has to be empty. If any token in `⟨expandable-tokens⟩` 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 `⟨expandable-tokens⟩` 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.

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

`\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.

`\exp:w` ★
`\exp_end_continue_f:nw` ★
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 a brace group then the braces are removed. If on the other hand it starts with space tokens then these space tokens are removed while searching for the argument. Thus such space tokens will not terminate the f-type expansion.

10 Internal functions and variables

`\l__exp_internal_tl`

The `\exp_` module has its private variables to temporarily store results of the argument expansion. This is done to avoid interference with other functions using temporary variables.

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

```
\::n \cs_set:Npn \exp_args:Ncof { \::c \::o \::f \::: }
\::N Internal forms for the base expansion types. These names do not conform to the general
\::P LATEX3 approach as this makes them more readily visible in the log and so forth.
\::c
\::o
\::f
\::x
\::v
\::V
\:::
```

Part VI

The `l3tl` package

Token lists

\TeX works with tokens, and \LaTeX 3 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 \TeX 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

```
\tl_new:N
\tl_new:c
```

```
\tl_new:N <tl var>
```

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.

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

```
\tl_const:Nn <tl var> {<token list>}
```

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>`.

```
\tl_clear:N
\tl_clear:c
\tl_gclear:N
\tl_gclear:c
```

```
\tl_clear:N <tl var>
```

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

<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 <code><tl var></code> 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 <code><tl var></code> empty.

<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 <code><tl var₁></code> equal to that of <code><tl var₂></code> .
<code>\tl_gset_eq:NN</code>	
<code>\tl_gset_eq:(cN Nc cc)</code>	

<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 <code><tl var₂></code> and <code><tl var₃></code> together and saves the result in
<code>\tl_gconcat:ccc</code>	<code><tl var₁></code> . The <code><tl var₂></code> is placed at the left side of the new token list.

New: 2012-05-18

<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 <code><tl var></code> is currently defined. This does not check that the <code><tl var></code> really is a token list variable.

New: 2012-03-03

2 Adding data to token list variables

<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>	

Sets `<tl var>` to contain `<tokens>`, removing any previous content from the variable.

<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>	

Appends `<tokens>` to the left side of the current content of `<tl var>`.

<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>	

Appends `<tokens>` to the right side of the current content of `<tl var>`.

3 Modifying token list variables

`\tl_replace_once:Nnn` `\tl_replace_once:Nnn <tl var> {(old tokens)} {(new tokens)}`
`\tl_replace_once:cnn`
`\tl_greplace_once:Nnn` Replaces the first (leftmost) occurrence of `<old tokens>` in the `<tl var>` with `<new tokens>`.
`\tl_greplace_once:cnn` `<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).

Updated: 2011-08-11

`\tl_replace_all:Nnn` `\tl_replace_all:Nnn <tl var> {(old tokens)} {(new tokens)}`
`\tl_replace_all:cnn`
`\tl_greplace_all:Nnn` Replaces all occurrences of `<old tokens>` in the `<tl var>` with `<new tokens>`. `<Old tokens>`
`\tl_greplace_all:cnn` 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).

Updated: 2011-08-11

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

Updated: 2011-08-11

`\tl_remove_all:Nn` `\tl_remove_all:Nn <tl var> {(tokens)}`
`\tl_remove_all:cn`
`\tl_gremove_all:Nn` Removes all occurrences of `<tokens>` from the `<tl var>`. `<Tokens>` cannot contain `{, }` or `#`
`\tl_gremove_all:cn` (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,

Updated: 2011-08-11

`\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).

```

\l_tl_set_rescan:Nnn \tl_set_rescan:Nnn <tl var> <{setup}> <{tokens}>
\l_tl_set_rescan:(Nno|Nnx|cnn|cno|cnx)
\l_tl_gset_rescan:Nnn
\l_tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)

```

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 $\backslash\text{tl_set_rescan:Nnn}$.) 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 $\backslash\text{tl_rescan:nn}$.

TeXhackers note: The $\langle tokens\rangle$ are first turned into a string (using $\backslash\text{tl_to_str:n}$). If the string contains one or more characters with character code $\backslash\text{newlinechar}$ (set equal to $\backslash\text{endlinechar}$ 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. Only the case of a single line is supported in LuaTeX because of a bug in this engine.

```

\l_tl_rescan:nn \tl_rescan:nn <{setup}> <{tokens}>

```

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 $\backslash\text{tl_rescan:nn}$.) 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 $\backslash\text{tl_set_rescan:Nnn}$, which is more robust than using $\backslash\text{tl_set:Nn}$ in the $\langle tokens\rangle$ argument of $\backslash\text{tl_rescan:nn}$.

TeXhackers note: The $\langle tokens\rangle$ are first turned into a string (using $\backslash\text{tl_to_str:n}$). If the string contains one or more characters with character code $\backslash\text{newlinechar}$ (set equal to $\backslash\text{endlinechar}$ 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. Only the case of a single line is supported in LuaTeX because of a bug in this engine.

5 Token list conditionals

```

\l_tl_if_blank_p:n * \tl_if_blank_p:n <{token list}>
\l_tl_if_blank_p:(V|o) * \tl_if_blank:nTF <{token list}> <{true code}> <{false code}>
\l_tl_if_blank:nTF *
\l_tl_if_blank:(V|o)TF *

```

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</code> \langle tl var \rangle
<code>\tl_if_empty_p:c</code> *	<code>\tl_if_empty:NTF</code> \langle tl var \rangle $\{$ \langle true code \rangle $\}$ $\{$ \langle false code \rangle $\}$
<code>\tl_if_empty:NTF</code> *	Tests if the \langle token list variable \rangle 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</code> $\{$ \langle token list \rangle $\}$
<code>\tl_if_empty_p:(V o)</code> *	<code>\tl_if_empty:nTF</code> $\{$ \langle token list \rangle $\}$ $\{$ \langle true code \rangle $\}$ $\{$ \langle false code \rangle $\}$
<code>\tl_if_empty:nTF</code> *	Tests if the \langle token list \rangle 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</code> \langle tl var $_1$ \rangle \langle tl var $_2$ \rangle
<code>\tl_if_eq_p:(Nc cN cc)</code> *	<code>\tl_if_eq:NNTF</code> \langle tl var $_1$ \rangle \langle tl var $_2$ \rangle $\{$ \langle true code \rangle $\}$ $\{$ \langle false code \rangle $\}$
<code>\tl_if_eq:NNTF</code> *	Compares the content of two \langle token list variables \rangle 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</code> $\{$ \langle token list $_1$ \rangle $\}$ $\{$ \langle token list $_2$ \rangle $\}$ $\{$ \langle true code \rangle $\}$ $\{$ \langle false code \rangle $\}$
	Tests if \langle token list $_1$ \rangle and \langle token list $_2$ \rangle 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</code> \langle tl var \rangle $\{$ \langle token list \rangle $\}$ $\{$ \langle true code \rangle $\}$ $\{$ \langle false code \rangle $\}$
<code>\tl_if_in:cnTF</code>	Tests if the \langle token list \rangle is found in the content of the \langle tl var \rangle . The \langle token list \rangle 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</code> $\{$ \langle token list $_1$ \rangle $\}$ $\{$ \langle token list $_2$ \rangle $\}$ $\{$ \langle true code \rangle $\}$ $\{$ \langle false code \rangle $\}$
<code>\tl_if_in:(Vn on no)TF</code>	Tests if \langle token list $_2$ \rangle is found inside \langle token list $_1$ \rangle . The \langle token list $_2$ \rangle 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</code> $\{$ \langle token list \rangle $\}$
<code>\tl_if_novalue:nTF</code> *	<code>\tl_if_novalue:nTF</code> $\{$ \langle token list \rangle $\}$ $\{$ \langle true code \rangle $\}$ $\{$ \langle false code \rangle $\}$
	Tests if the \langle token list \rangle is exactly equal to the special <code>\c_novalue_tl</code> marker. This function is indented to allow construction of flexible document interface structures in which missing optional arguments are detected.

New: 2017-11-14

```

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

```

```

\Tl_if_single_p:n * \Tl_if_single_p:n {<token list>}
\Tl_if_single:nTF * \Tl_if_single:nTF {<token list>} {<true code>} {<false code>}
Updated: 2011-08-13 Tests if the <token list> has exactly one item, i.e. 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 \Tl_count:n.

```

```

\Tl_case:Nn * \Tl_case:NnTF <test token list variable>
\Tl_case:cn * {
\Tl_case:NnTF * <token list variable case1> {<code case1>}
\Tl_case:cnTF * <token list variable case2> {<code case2>}
New: 2013-07-24 ...
<token list variable casen> {<code casen>}
}
{<true code>}
{<false code>}

```

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

```

\Tl_map_function:NN * \Tl_map_function:NN <tl var> <function>
\Tl_map_function:cN *
Updated: 2012-06-29

```

Applies *<function>* to every *<item>* in the *<tl var>*. The *<function>* receives one argument for each iteration. This may be a number of tokens if the *<item>* was stored within braces. Hence the *<function>* should anticipate receiving n-type arguments. See also `\Tl_map_function:nN`.

```

\Tl_map_function:nN * \Tl_map_function:nN {<token list>} <function>
Updated: 2012-06-29

```

Applies *<function>* to every *<item>* in the *<token list>*, The *<function>* receives one argument for each iteration. This may be a number of tokens if the *<item>* was stored within braces. Hence the *<function>* should anticipate receiving n-type arguments. See also `\Tl_map_function:NN`.

```

\Tl_map_inline:Nn * \Tl_map_inline:Nn <tl var> {<inline function>}
\Tl_map_inline:cn *
Updated: 2012-06-29

```

Applies the *<inline function>* to every *<item>* stored within the *<tl var>*. The *<inline function>* should consist of code which receives the *<item>* as #1. One in line mapping can be nested inside another. See also `\Tl_map_function:NN`.

`\tl_map_inline:nn`

Updated: 2012-06-29

`\tl_map_inline:nn` $\langle token\ list \rangle$ $\langle inline\ function \rangle$

Applies the $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle token\ list \rangle$. The $\langle inline\ function \rangle$ should consist of code which receives the $\langle item \rangle$ as #1. One in line mapping can be nested inside another. See also `\tl_map_function:nn`.

`\tl_map_variable:NNn`

`\tl_map_variable:cNn`

Updated: 2012-06-29

`\tl_map_variable:NNn` $\langle tl\ var \rangle$ $\langle variable \rangle$ $\langle function \rangle$

Applies the $\langle function \rangle$ to every $\langle item \rangle$ stored within the $\langle tl\ var \rangle$. The $\langle function \rangle$ should consist of code which receives the $\langle item \rangle$ stored in the $\langle variable \rangle$. One variable mapping can be nested inside another. See also `\tl_map_inline:Nn`.

`\tl_map_variable:nNn`

Updated: 2012-06-29

`\tl_map_variable:nNn` $\langle token\ list \rangle$ $\langle variable \rangle$ $\langle function \rangle$

Applies the $\langle function \rangle$ to every $\langle item \rangle$ stored within the $\langle token\ list \rangle$. The $\langle function \rangle$ should consist of code which receives the $\langle item \rangle$ stored in the $\langle variable \rangle$. One variable mapping can be nested inside another. See also `\tl_map_inline:nn`.

`\tl_map_break:` ☆

Updated: 2012-06-29

`\tl_map_break:`

Used to terminate a `\tl_map...` function before all entries in the $\langle token\ list\ variable \rangle$ have been processed. 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: }
  % Do something useful
}
```

See also `\tl_map_break:n`. 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 by the internal macro `__prg_break_point:Nn` before the $\langle tokens \rangle$ are inserted into the input stream. This depends on the design of the mapping function.

`\tl_map_break:n` ☆

Updated: 2012-06-29

`\tl_map_break:n` {<tokens>}

Used to terminate a `\tl_map_...` function before all entries in the <token list variable> have been processed, inserting the <tokens> 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 { <tokens> } }
  % 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 by the internal macro `__prg_break_point:Nn` before the <tokens> are 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).

TeXhackers note: 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`.

`\tl_use:N` ★ `\tl_use:N <tl var>`

`\tl_use:c` ★

Recovers the content of a *<tl 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 *<tl var>* directly without an accessor function.

8 Working with the content of token lists

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

`\tl_count:(V|o)` ★

New: 2012-05-13

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

`\tl_count:N` ★ `\tl_count:N <tl var>`

`\tl_count:c` ★

New: 2012-05-13

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

`\tl_reverse:n` ★ `\tl_reverse:n {(token list)}`

`\tl_reverse:(V|o)` ★

Updated: 2012-01-08

Reverses the order of the *<items>* in the *<token list>*, so that *<item₁><item₂><item₃>...<item_n>* becomes *<item_n>...<item₃><item₂><item₁>*. This process preserves unprotected space within the *<token list>*. 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.

`\tl_reverse:N` `\tl_reverse:N <tl var>`

`\tl_reverse:c`

`\tl_greverse:N`

`\tl_greverse:c`

Updated: 2012-01-08

Reverses the order of the *<items>* stored in *<tl var>*, so that *<item₁><item₂><item₃>...<item_n>* becomes *<item_n>...<item₃><item₂><item₁>*. This process preserves unprotected spaces within the *<token list variable>*. 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`.

`\tl_reverse_items:n` ★ `\tl_reverse_items:n {(token list)}`

New: 2012-01-08

Reverses the order of the *<items>* stored in *<tl var>*, so that *{<item₁>}{<item₂>}{<item₃>}...{<item_n>}* becomes *{<item_n>}...{<item₃>}{<item₂>}{<item₁>}*. This process removes any unprotected space within the *<token list>*. 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</code> $\langle token list \rangle$
<small>New: 2011-07-09 Updated: 2012-06-25</small>	Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the $\langle token list \rangle$ and leaves the result in the input stream.

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

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

<code>\tl_sort:nN</code> ★	<code>\tl_sort:nN</code> $\langle token list \rangle$ $\langle conditional \rangle$
<small>New: 2017-02-06</small>	Sorts the items in the $\langle token list \rangle$, using the $\langle conditional \rangle$ to compare items, and leaves the result in the input stream. The $\langle conditional \rangle$ 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.

`\tl_head:N` ★
`\tl_head:n` ★
`\tl_head:(V|v|f)` ★

Updated: 2012-09-09

`\tl_head:n` $\langle token list \rangle$

Leaves in the input stream the first $\langle item \rangle$ in the $\langle token list \rangle$, discarding the rest of the $\langle token list \rangle$. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

`\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 $\langle token list \rangle$ (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.

`\tl_head:w` ★

`\tl_head:w` $\langle token list \rangle$ { } `\q_stop`

Leaves in the input stream the first $\langle item \rangle$ in the $\langle token list \rangle$, discarding the rest of the $\langle token list \rangle$. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank $\langle token list \rangle$ (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.

`\tl_tail:N` ★
`\tl_tail:n` ★
`\tl_tail:(V|v|f)` ★

Updated: 2012-09-01

`\tl_tail:n` $\langle token list \rangle$

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first $\langle item \rangle$ in the $\langle token list \rangle$, and leaves the remaining tokens in the input stream. Thus for example

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

and

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

both leave `▯{bc}d` in the input stream. A blank $\langle token list \rangle$ (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.

```

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

```

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

```

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**.

```

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

```

\tl_if_head_is_group_p:n * \tl_if_head_is_group_p:n {<token list>}
\tl_if_head_is_group:nTF * \tl_if_head_is_group:nTF {<token list>} {<true code>} {<false 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.

```

\tl_if_head_is_N_type_p:n * \tl_if_head_is_N_type_p:n {<token list>}
\tl_if_head_is_N_type:nTF * \tl_if_head_is_N_type:nTF {<token list>} {<true code>} {<false 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.

```

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

`\tl_item:nn` * `\tl_item:nn {⟨token list⟩} {⟨integer expression⟩}`

`\tl_item:Nn` *

`\tl_item:cn` *

New: 2014-07-17

Indexing items in the *⟨token list⟩* from 1 on the left, this function evaluates the *⟨integer expression⟩* and leaves the appropriate item from the *⟨token list⟩* in the input stream. If the *⟨integer expression⟩* 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.

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.

11 Viewing token lists

`\tl_show:N` `\tl_show:N ⟨tl var⟩`

`\tl_show:c`

Updated: 2015-08-01

Displays the content of the *⟨tl var⟩* on the terminal.

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

`\tl_show:n` `\tl_show:n {⟨token list⟩}`

Updated: 2015-08-07

Displays the *⟨token list⟩* on the terminal.

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

`\tl_log:N` `\tl_log:N ⟨tl var⟩`

`\tl_log:c`

New: 2014-08-22

Updated: 2015-08-01

Writes the content of the *⟨tl var⟩* in the log file. See also `\tl_show:N` which displays the result in the terminal.

`\tl_log:n` `\tl_log:n {⟨token list⟩}`

New: 2014-08-22

Updated: 2015-08-07

Writes the *⟨token list⟩* in the log file. See also `\tl_show:n` which displays the result in the terminal.

12 Constant token lists

`\c_empty_tl` Constant that is always empty.

`\c_novalue_tl`
New: 2017-11-14

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

`\c_space_tl`

An explicit space character contained in a token list (compare this with `\c_space_token`). For use where an explicit space is required.

13 Scratch token lists

`\l_tmpa_tl`
`\l_tmpb_tl`

Scratch token lists 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_tl`
`\g_tmpb_tl`

Scratch token lists 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.

14 Internal functions

`_tl_trim_spaces:n`

```
\_tl_trim_spaces:n { \q_mark <token list> } {<continuation>}
```

This function removes all leading and trailing explicit space characters from the *<token list>*, and expands to the *<continuation>*, followed by a brace group containing `\use_none:n \q_mark <trimmed token list>`. For instance, `\tl_trim_spaces:n` is implemented by taking the *<continuation>* to be `\exp_not:o`, and the `o`-type expansion removes the `\q_mark`. This function is also used in `l3clist` and `l3candidates`.

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` $\langle str\ var \rangle$

Creates a new $\langle str\ var \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle str\ var \rangle$ is initially empty.

`\str_const:Nn`

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

New: 2015-09-18

`\str_const:Nn` $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$

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

<code>\str_clear:N</code>	<code>\str_clear:N</code> $\langle str\ var \rangle$
<code>\str_clear:c</code>	
<code>\str_gclear:N</code>	Clears the content of the $\langle str\ var \rangle$.
<code>\str_gclear:c</code>	

New: 2015-09-18

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

New: 2015-09-18

<code>\str_set_eq:NN</code>	<code>\str_set_eq:NN</code> $\langle str\ var_1 \rangle$ $\langle str\ var_2 \rangle$
<code>\str_set_eq:(cN Nc cc)</code>	
<code>\str_gset_eq:NN</code>	Sets the content of $\langle str\ var_1 \rangle$ equal to that of $\langle str\ var_2 \rangle$.
<code>\str_gset_eq:(cN Nc cc)</code>	

New: 2015-09-18

<code>\str_concat:NNN</code>	<code>\str_concat:NNN</code> $\langle str\ var_1 \rangle$ $\langle str\ var_2 \rangle$ $\langle str\ var_3 \rangle$
<code>\str_concat:ccc</code>	
<code>\str_gconcat:NNN</code>	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.
<code>\str_gconcat:ccc</code>	

New: 2017-10-08

2 Adding data to string variables

<code>\str_set:Nn</code>	<code>\str_set:Nn</code> $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$
<code>\str_set:(Nx cn cx)</code>	
<code>\str_gset:Nn</code>	Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$, and stores the result in $\langle str\ var \rangle$.
<code>\str_gset:(Nx cn cx)</code>	

New: 2015-09-18

<code>\str_put_left:Nn</code>	<code>\str_put_left:Nn</code> $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$
<code>\str_put_left:(Nx cn cx)</code>	
<code>\str_gput_left:Nn</code>	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.
<code>\str_gput_left:(Nx cn cx)</code>	

New: 2015-09-18

<code>\str_put_right:Nn</code>	<code>\str_put_right:Nn</code> $\langle str\ var \rangle$ $\{ \langle token\ list \rangle \}$
<code>\str_put_right:(Nx cn cx)</code>	
<code>\str_gput_right:Nn</code>	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.
<code>\str_gput_right:(Nx cn cx)</code>	

New: 2015-09-18

2.1 Modifying string variables

`\str_replace_once:Nnn` `\str_replace_once:Nnn <str var> {<old>} {<new>}`
`\str_replace_once:cn` Converts the `<old>` and `<new>` token lists to strings, then replaces the first (leftmost)
`\str_greplace_once:Nnn` occurrence of `<old string>` in the `<str var>` with `<new string>`.
`\str_greplace_once:cn`

New: 2017-10-08

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

New: 2017-10-08

`\str_remove_once:Nn` `\str_remove_once:Nn <str var> {<token list>}`
`\str_remove_once:cn` Converts the `<token list>` to a `<string>` then removes the first (leftmost) occurrence of
`\str_gremove_once:Nn` `<string>` from the `<str var>`.
`\str_gremove_once:cn`

New: 2017-10-08

`\str_remove_all:Nn` `\str_remove_all:Nn <str var> {<token list>}`
`\str_remove_all:cn` Converts the `<token list>` to a `<string>` then removes all occurrences of `<string>` from the
`\str_gremove_all:Nn` `<str var>`. As this function operates from left to right, the pattern `<string>` may remain
`\str_gremove_all:cn` after the removal, for instance,

New: 2017-10-08

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

results in `\l_tmpa_str` containing `abcd`.

2.2 String conditionals

`\str_if_exist_p:N *` `\str_if_exist_p:N <str var>`
`\str_if_exist_p:c *` `\str_if_exist:NTF <str var> {<true code>} {<false code>}`
`\str_if_exist:NTF *` Tests whether the `<str var>` is currently defined. This does not check that the `<str var>`
`\str_if_exist:cTF *` really is a string.

New: 2015-09-18

`\str_if_empty_p:N *` `\str_if_empty_p:N <str var>`
`\str_if_empty_p:c *` `\str_if_empty:NTF <str var> {<true code>} {<false code>}`
`\str_if_empty:NTF *` Tests if the `<string variable>` is entirely empty (*i.e.* contains no characters at all).
`\str_if_empty:cTF *`

New: 2015-09-18

<code>\str_if_eq_p:NN</code>	*	<code>\str_if_eq_p:NN</code>	$\langle str\ var_1 \rangle$	$\langle str\ var_2 \rangle$	
<code>\str_if_eq_p:(Nc cN cc)</code>	*	<code>\str_if_eq:NNTF</code>	$\langle str\ var_1 \rangle$	$\langle str\ var_2 \rangle$	$\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
<code>\str_if_eq:NNTF</code>	*				
<code>\str_if_eq:(Nc cN cc)TF</code>	*	Compares the content of two $\langle str\ variables \rangle$ 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</code>	$\{\langle t_1 \rangle\}$	$\{\langle t_2 \rangle\}$	
<code>\str_if_eq_p:(Vn on no nV VV)</code>	*	<code>\str_if_eq:nnTF</code>	$\{\langle t_1 \rangle\}$	$\{\langle t_2 \rangle\}$	$\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
<code>\str_if_eq:nnTF</code>	*				
<code>\str_if_eq:(Vn on no nV VV)TF</code>	*				

Compares the two $\langle token\ lists \rangle$ on a character by character basis (namely after converting them to strings), and is true if the two $\langle strings \rangle$ 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</code>	$\{\langle t_1 \rangle\}$	$\{\langle t_2 \rangle\}$	
<code>\str_if_eq_x:nnTF</code>	*	<code>\str_if_eq_x:nnTF</code>	$\{\langle t_1 \rangle\}$	$\{\langle t_2 \rangle\}$	$\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$

New: 2012-06-05

Fully expands the two $\langle token\ lists \rangle$ and converts them to $\langle strings \rangle$, then compares these on a character by character basis: it is true if the two $\langle strings \rangle$ 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</code>	$\langle str\ var \rangle$	$\{\langle token\ list \rangle\}$	$\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
------------------------------	--	------------------------------	----------------------------	-----------------------------------	--

`\str_if_in:cnTF`

New: 2017-10-08

Converts the $\langle token\ list \rangle$ to a $\langle string \rangle$ and tests if that $\langle string \rangle$ is found in the content of the $\langle str\ var \rangle$.

<code>\str_if_in:nnTF</code>		<code>\str_if_in:nnTF</code>	$\langle t_1 \rangle$	$\{\langle t_2 \rangle\}$	$\{\langle true\ code \rangle\} \{\langle false\ code \rangle\}$
------------------------------	--	------------------------------	-----------------------	---------------------------	--

New: 2017-10-08

Converts both $\langle token\ lists \rangle$ to $\langle strings \rangle$ and tests whether $\langle string_2 \rangle$ is found inside $\langle string_1 \rangle$.

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

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

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.

3 Mapping to strings

<code>\str_map_function:NN</code>	☆	<code>\str_map_function:NN</code>	⟨ <i>str var</i> ⟩ ⟨ <i>function</i> ⟩
<code>\str_map_function:cN</code>	☆		Applies ⟨ <i>function</i> ⟩ to every ⟨ <i>character</i> ⟩ in the ⟨ <i>str var</i> ⟩. See also <code>\str_map_function:nN</code> .

New: 2017-10-08

<code>\str_map_function:nN</code>	☆	<code>\str_map_function:nN</code>	{⟨ <i>token list</i> ⟩} ⟨ <i>function</i> ⟩
			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> ⟩. See also <code>\str_map_function:NN</code> .

New: 2017-10-08

<code>\str_map_inline:Nn</code>	<code>\str_map_inline:Nn <str var> {(inline function)}</code>
<code>\str_map_inline:cn</code>	Applies the <i><inline function></i> to every <i><character></i> in the <i><str var></i> . The <i><inline function></i> should consist of code which receives the <i><character></i> as #1. One in line mapping can be nested inside another. See also <code>\str_map_function:NN</code> .
New: 2017-10-08	
<code>\str_map_inline:nn</code>	<code>\str_map_inline:nn {(token list)} {(inline function)}</code>
New: 2017-10-08	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> . The <i><inline function></i> should consist of code which receives the <i><character></i> as #1. One in line mapping can be nested inside another. See also <code>\str_map_function:NN</code> .
<code>\str_map_variable:NNn</code>	<code>\str_map_variable:NNn <str var> <variable> {(function)}</code>
<code>\str_map_variable:cNn</code>	Applies the <i><function></i> to every <i><character></i> in the <i><str var></i> . The <i><function></i> should consist of code which receives the <i><character></i> stored in the <i><variable></i> . One variable mapping can be nested inside another. See also <code>\str_map_inline:Nn</code> .
New: 2017-10-08	
<code>\str_map_variable:nNn</code>	<code>\str_map_variable:nNn {(token list)} <variable> {(function)}</code>
New: 2017-10-08	Converts the <i><token list></i> to a <i><string></i> then applies the <i><function></i> to every <i><character></i> in the <i><string></i> . The <i><function></i> should consist of code which receives the <i><character></i> stored in the <i><variable></i> . One variable mapping can be nested inside another. See also <code>\str_map_inline:Nn</code> .
<code>\str_map_break: ☆</code>	<code>\str_map_break:</code>
New: 2017-10-08	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>
	See also <code>\str_map_break:n</code> . Use outside of a <code>\str_map...</code> scenario leads to low level TeX errors.
	TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro <code>__prg_break_point:Nn</code> before continuing with the code that follows the loop. This depends on the design of the mapping function.

`\str_map_break:n` ☆

New: 2017-10-08

`\str_map_break:n` $\{(tokens)\}$

Used to terminate a `\str_map_...` function before all characters in the $\langle string \rangle$ have been processed, inserting the $\langle tokens \rangle$ 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 { <tokens> } }
  % 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 by the internal macro `__prg_break_point:Nn` before the $\langle tokens \rangle$ are inserted into the input stream. This depends on the design of the mapping function.

4 Working with the content of strings

`\str_use:N` ☆

`\str_use:c` ☆

New: 2015-09-18

`\str_use:N` $\langle str var \rangle$

Recovers the content of a $\langle str var \rangle$ 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 $\langle str \rangle$ 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 $\langle token list \rangle$, 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 $\langle token list \rangle$, as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

<code>\str_head:N</code>	★	<code>\str_head:n</code> $\langle\{token\ list\}\rangle$
<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</code> $\langle\{token\ list\}\rangle$
<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</code> $\langle\{token\ list\}\rangle$ $\langle\{integer\ expression\}\rangle$
<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 *⟨token list⟩* to a *⟨string⟩*, and leaves in the input stream the characters from the *⟨start index⟩* to the *⟨end index⟩* inclusive. Positive *⟨indices⟩* are counted from the start of the string, 1 being the first character, and negative *⟨indices⟩* are counted from the end of the string, -1 being the last character. If either of *⟨start index⟩* or *⟨end index⟩* 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 *⟨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 { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:x { \str_range:nnn { abcdef } { -1 } { -4 } }

```

both yield empty strings.

5 String manipulation

```
\str_lower_case:n * \str_lower_case:n {<tokens>}
\str_lower_case:f * \str_upper_case:n {<tokens>}
\str_upper_case:n *
\str_upper_case:f *
```

New: 2015-03-01

Converts the input $\langle tokens \rangle$ to their string representation, as described for $\backslash\text{tl_to_str:n}$, 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 $\backslash\text{str_fold_case:n}$ for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the $\backslash\text{tl_lower_case:n(n)}$, $\backslash\text{tl_upper_case:n(n)}$ and $\backslash\text{tl_mixed_case:n(n)}$ 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 $\backslash\text{str_fold_case:n}$ 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_YT_EX and LuaT_EX, subject only to the fact that X_YT_EX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with $\backslash\text{tl_to_str:n}$.

`\str_fold_case:n` ★

`\str_fold_case:V` ★

New: 2014-06-19

Updated: 2016-03-07

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

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.* `SS` folds 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 `XƎTeX` and `LuaTeX`, subject only to the fact that `XƎTeX` in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

6 Viewing strings

`\str_show:N`

`\str_show:c`

`\str_show:n`

New: 2015-09-18

`\str_show:N` $\langle str var \rangle$

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

7 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`

New: 2015-09-19

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

8 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 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_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 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.

8.1 Internal string functions

`_str_if_eq_x:nn` ★ `_str_if_eq_x:nn` $\{\langle t_1 \rangle\}$ $\{\langle t_2 \rangle\}$

Compares the full expansion of two $\langle token\ lists \rangle$ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Leaves 0 in the input stream if the condition is true, and +1 or -1 otherwise.

`_str_if_eq_x_return:nn` `_str_if_eq_x_return:nn` $\{\langle t_1 \rangle\}$ $\{\langle t_2 \rangle\}$

Compares the full expansion of two $\langle token\ lists \rangle$ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Either `\prg_return_true:` or `\prg_return_false:` is then left in the input stream. This is a version of `\str_if_eq_x:nnTF` coded for speed.

`_str_to_other:n` ★ `_str_to_other:n` $\{\langle token\ list \rangle\}$

Converts the $\langle token\ list \rangle$ to a $\langle other\ string \rangle$, 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.

`_str_to_other_fast:n` ☆ `_str_to_other_fast:n` $\{\langle token\ list\rangle\}$
Same behaviour `_str_to_other:n` but only restricted-expandable. It takes a time linear in the character count of the string. It is used for `\iow_wrap:nnnN`.

`_str_count:n` ☆ `_str_count:n` $\{\langle other\ string\rangle\}$
This function expects an argument that is entirely made of characters with category “other”, as produced by `_str_to_other:n`. It leaves in the input stream the number of character tokens in the $\langle other\ string\rangle$, faster than the analogous `\str_count:n` function.

`_str_range:nnn` ☆ `_str_range:nnn` $\{\langle other\ string\rangle\}$ $\{\langle start\ index\rangle\}$ $\{\langle end\ index\rangle\}$
Identical to `\str_range:nnn` except that the first argument is expected to be entirely made of characters with category “other”, as produced by `_str_to_other:n`, and the result is also an $\langle other\ string\rangle$.

Part VIII

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

`\seq_new:N`
`\seq_new:c`

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

Creates a new $\langle sequence \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle sequence \rangle$ initially contains no items.

`\seq_clear:N`
`\seq_clear:c`
`\seq_gclear:N`
`\seq_gclear:c`

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

Clears all items from the $\langle sequence \rangle$.

`\seq_clear_new:N`
`\seq_clear_new:c`
`\seq_gclear_new:N`
`\seq_gclear_new:c`

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

Ensures that the $\langle sequence \rangle$ exists globally by applying `\seq_new:N` if necessary, then applies `\seq_(g)clear:N` to leave the $\langle sequence \rangle$ empty.

`\seq_set_eq:NN`
`\seq_set_eq:(cN|Nc|cc)`
`\seq_gset_eq:NN`
`\seq_gset_eq:(cN|Nc|cc)`

`\seq_set_eq:NN` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$

Sets the content of $\langle sequence_1 \rangle$ equal to that of $\langle sequence_2 \rangle$.

`\seq_set_from_clist:NN`
`\seq_set_from_clist:(cN|Nc|cc)`
`\seq_set_from_clist:Nn`
`\seq_set_from_clist:cn`
`\seq_gset_from_clist:NN`
`\seq_gset_from_clist:(cN|Nc|cc)`
`\seq_gset_from_clist:Nn`
`\seq_gset_from_clist:cn`

`\seq_set_from_clist:NN` $\langle sequence \rangle$ $\langle comma-list \rangle$

New: 2014-07-17

Converts the data in the $\langle comma list \rangle$ into a $\langle sequence \rangle$: the original $\langle comma list \rangle$ is unchanged.

<code>\seq_set_split:Nnn</code>	<code>\seq_set_split:Nnn <sequence> <{delimiter}> <{token list}></code>
<code>\seq_set_split:NnV</code>	Splits the <i><token list></i> into <i><items></i> separated by <i><delimiter></i> , and assigns the result to the <i><sequence></i> . Spaces on both sides of each <i><item></i> are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of <code>\l3clist</code> functions. Empty <i><items></i> are preserved by <code>\seq_set_split:Nnn</code> , and can be removed afterwards using <code>\seq_remove_all:Nn <sequence> <{}></code> . The <i><delimiter></i> may not contain <code>{, }</code> or <code>#</code> (assuming <code>TeX</code> 's normal category code régime). If the <i><delimiter></i> is empty, the <i><token list></i> is split into <i><items></i> as a <i><token list></i> .
<code>\seq_gset_split:Nnn</code>	
<code>\seq_gset_split:NnV</code>	
New: 2011-08-15 Updated: 2012-07-02	

<code>\seq_concat:NNN</code>	<code>\seq_concat:NNN <sequence₁> <sequence₂> <sequence₃></code>
<code>\seq_concat:ccc</code>	Concatenates the content of <i><sequence₂></i> and <i><sequence₃></i> together and saves the result in <i><sequence₁></i> . The items in <i><sequence₂></i> are placed at the left side of the new sequence.
<code>\seq_gconcat:NNN</code>	
<code>\seq_gconcat:ccc</code>	

<code>\seq_if_exist_p:N</code> *	<code>\seq_if_exist_p:N <sequence></code>
<code>\seq_if_exist_p:c</code> *	<code>\seq_if_exist:NTF <sequence> <{true code}> <{false code}></code>
<code>\seq_if_exist:NTF</code> *	Tests whether the <i><sequence></i> is currently defined. This does not check that the <i><sequence></i> really is a sequence variable.
<code>\seq_if_exist:cTF</code> *	
New: 2012-03-03	

2 Appending data to sequences

<code>\seq_put_left:Nn</code>	<code>\seq_put_left:Nn <sequence> <{item}></code>
<code>\seq_put_left:(NV Nv No Nx cn cV cv co cx)</code>	
<code>\seq_gput_left:Nn</code>	
<code>\seq_gput_left:(NV Nv No Nx cn cV cv co cx)</code>	

Appends the *<item>* to the left of the *<sequence>*.

<code>\seq_put_right:Nn</code>	<code>\seq_put_right:Nn <sequence> <{item}></code>
<code>\seq_put_right:(NV Nv No Nx cn cV cv co cx)</code>	
<code>\seq_gput_right:Nn</code>	
<code>\seq_gput_right:(NV Nv No Nx cn cV cv co cx)</code>	

Appends the *<item>* to the right of the *<sequence>*.

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 *<token list variable>* used with `\tl_set:Nn` and *never* `\tl_gset:Nn`.

<code>\seq_get_left:NN</code>	<code>\seq_get_left:NN <sequence> <token list variable></code>
<code>\seq_get_left:cN</code>	Stores the left-most 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. 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	

`\seq_get_right:NN` `\seq_get_right:NN` $\langle sequence \rangle$ $\langle token list variable \rangle$
`\seq_get_right:cN`
Updated: 2012-05-19

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 `\q_no_value`.

`\seq_pop_left:NN` `\seq_pop_left:NN` $\langle sequence \rangle$ $\langle token list variable \rangle$
`\seq_pop_left:cN`
Updated: 2012-05-14

Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, *i.e.* 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 `\q_no_value`.

`\seq_gpop_left:NN` `\seq_gpop_left:NN` $\langle sequence \rangle$ $\langle token list variable \rangle$
`\seq_gpop_left:cN`
Updated: 2012-05-14

Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, *i.e.* 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 `\q_no_value`.

`\seq_pop_right:NN` `\seq_pop_right:NN` $\langle sequence \rangle$ $\langle token list variable \rangle$
`\seq_pop_right:cN`
Updated: 2012-05-19

Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, *i.e.* 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 `\q_no_value`.

`\seq_gpop_right:NN` `\seq_gpop_right:NN` $\langle sequence \rangle$ $\langle token list variable \rangle$
`\seq_gpop_right:cN`
Updated: 2012-05-19

Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token list variable \rangle$, *i.e.* 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 `\q_no_value`.

`\seq_item:Nn` \star `\seq_item:Nn` $\langle sequence \rangle$ $\{ \langle integer expression \rangle \}$
`\seq_item:cN` \star
New: 2014-07-17

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 `\seq_count:N`) 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.

`\seq_get_left:NNTF`

`\seq_get_left:cNTF`

New: 2012-05-14

Updated: 2012-05-19

`\seq_get_left:NNTF` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle sequence \rangle$ 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 $\langle sequence \rangle$ is non-empty, stores the left-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from a $\langle sequence \rangle$. The $\langle token list variable \rangle$ is assigned locally.

`\seq_get_right:NNTF`

`\seq_get_right:cNTF`

New: 2012-05-19

`\seq_get_right:NNTF` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle sequence \rangle$ 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 $\langle sequence \rangle$ is non-empty, stores the right-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$ without removing it from a $\langle sequence \rangle$. The $\langle token list variable \rangle$ is assigned locally.

`\seq_pop_left:NNTF`

`\seq_pop_left:cNTF`

New: 2012-05-14

Updated: 2012-05-19

`\seq_pop_left:NNTF` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle sequence \rangle$ 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 $\langle sequence \rangle$ is non-empty, pops the left-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from a $\langle sequence \rangle$. Both the $\langle sequence \rangle$ and the $\langle token list variable \rangle$ are assigned locally.

`\seq_gpop_left:NNTF`

`\seq_gpop_left:cNTF`

New: 2012-05-14

Updated: 2012-05-19

`\seq_gpop_left:NNTF` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle sequence \rangle$ 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 $\langle sequence \rangle$ is non-empty, pops the left-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from a $\langle sequence \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token list variable \rangle$ is assigned locally.

`\seq_pop_right:NNTF`

`\seq_pop_right:cNTF`

New: 2012-05-19

`\seq_pop_right:NNTF` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle sequence \rangle$ 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 $\langle sequence \rangle$ is non-empty, pops the right-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from a $\langle sequence \rangle$. Both the $\langle sequence \rangle$ and the $\langle token list variable \rangle$ are assigned locally.

`\seq_gpop_right:NNTF`

`\seq_gpop_right:cNTF`

New: 2012-05-19

`\seq_gpop_right:NNTF` $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

If the $\langle sequence \rangle$ 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 $\langle sequence \rangle$ is non-empty, pops the right-most item from a $\langle sequence \rangle$ in the $\langle token list variable \rangle$, *i.e.* removes the item from a $\langle sequence \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token list variable \rangle$ is assigned locally.

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.

<code>\seq_remove_duplicates:N</code>	<code>\seq_remove_duplicates:N <sequence></code>
<code>\seq_remove_duplicates:c</code>	
<code>\seq_gremove_duplicates:N</code>	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 <code>\tl_if_eq:nnTF</code> .
<code>\seq_gremove_duplicates:c</code>	

TeXhackers 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.

<code>\seq_remove_all:Nn</code>	<code>\seq_remove_all:Nn <sequence> {<item>}</code>
<code>\seq_remove_all:cn</code>	
<code>\seq_gremove_all:Nn</code>	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 <code>\tl_if_eq:nnTF</code> .
<code>\seq_gremove_all:cn</code>	

<code>\seq_reverse:N</code>	<code>\seq_reverse:N <sequence></code>
<code>\seq_reverse:c</code>	
<code>\seq_greverse:N</code>	Reverses the order of the items stored in the $\langle sequence \rangle$.
<code>\seq_greverse:c</code>	

New: 2014-07-18

<code>\seq_sort:Nn</code>	<code>\seq_sort:Nn <sequence> {<comparison code>}</code>
<code>\seq_sort:cn</code>	
<code>\seq_gsort:Nn</code>	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.
<code>\seq_gsort:cn</code>	

New: 2017-02-06

6 Sequence conditionals

<code>\seq_if_empty_p:N</code> ★	<code>\seq_if_empty_p:N <sequence></code>
<code>\seq_if_empty_p:c</code> ★	<code>\seq_if_empty:NNTF <sequence> {<true code>} {<false code>}</code>
<code>\seq_if_empty:NNTF</code> ★	Tests if the $\langle sequence \rangle$ is empty (containing no items).
<code>\seq_if_empty:cTF</code> ★	

<code>\seq_if_in:NnTF</code>	<code>\seq_if_in:NnTF <sequence> {<item>} {<true code>} {<false code>}</code>
<code>\seq_if_in:(NV Nv No Nx cn cV cv co cx)TF</code>	

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

7 Mapping to sequences

<code>\seq_map_function:NN</code> ★	<code>\seq_map_function:NN <sequence> <function></code>
<code>\seq_map_function:cN</code> ★	
Updated: 2012-06-29	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 <code>\seq_map_inline:Nn</code> is faster than <code>\seq_map_function:NN</code> for sequences with more than about 10 items. One mapping may be nested inside another.

`\seq_map_inline:Nn` `\seq_map_inline:Nn <sequence> {<inline function>}`

`\seq_map_inline:cn`

Updated: 2012-06-29

Applies *<inline function>* to every *<item>* stored within the *<sequence>*. The *<inline function>* should consist of code which will receive the *<item>* as #1. One in line mapping can be nested inside another. The *<items>* are returned from left to right.

`\seq_map_variable:NNn`

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

Updated: 2012-06-29

`\seq_map_variable:NNn <sequence> <tl var.> {<function using tl var.>}`

Stores each entry in the *<sequence>* in turn in the *<tl var.>* and applies the *<function using tl var.>* The *<function>* will usually consist of code making use of the *<tl var.>*, but this is not enforced. The assignments to *<tl var.>* are local, and one variable mapping can be nested inside another. The *<items>* are returned from left to right.

`\seq_map_break: ☆`

`\seq_map_break:`

Updated: 2012-06-29

Used to terminate a `\seq_map...` function before all entries in the *<sequence>* 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 T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` 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` $\{(tokens)\}$

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ have been processed, inserting the $\langle tokens \rangle$ 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 { <tokens> } }
  {
    % 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 by the internal macro `__prg_break_point:Nn` before the $\langle tokens \rangle$ are 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` $\langle sequence \rangle$

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` $\langle seq var \rangle$ $\{\langle separator between two \rangle\}$

$\{\langle separator between more than two \rangle\}$ $\{\langle separator between final two \rangle\}$

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.

`\seq_use:Nn` *★*

`\seq_use:cn` *★*

New: 2013-05-26

`\seq_use:Nn` $\langle seq\ var\rangle$ $\{\langle separator\rangle\}$

Places the contents of the $\langle seq\ var\rangle$ in the input stream, with the $\langle separator\rangle$ between the items. If the sequence has a single item, it is placed in the input stream with no $\langle separator\rangle$, 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:Nn \l_tmpa_seq { ~and~ }
```

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

T_EXhackers 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.

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.

`\seq_get:NN`

`\seq_get:cN`

Updated: 2012-05-14

`\seq_get:NN` $\langle sequence\rangle$ $\langle token\ list\ variable\rangle$

Reads the top item from a $\langle sequence\rangle$ into 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`.

`\seq_pop:NN`

`\seq_pop:cN`

Updated: 2012-05-14

`\seq_pop:NN` $\langle sequence\rangle$ $\langle token\ list\ variable\rangle$

Pops the top item from a $\langle sequence\rangle$ into 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 `\q_no_value`.

`\seq_gpop:NN`

`\seq_gpop:cN`

Updated: 2012-05-14

`\seq_gpop:NN` $\langle sequence\rangle$ $\langle token\ list\ variable\rangle$

Pops the top item from a $\langle sequence\rangle$ into the $\langle token\ list\ variable\rangle$. The $\langle sequence\rangle$ is modified globally, while 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`.

`\seq_get:NNTF`

`\seq_get:cNTF`

New: 2012-05-14

Updated: 2012-05-19

`\seq_get:NNTF` $\langle sequence\rangle$ $\langle token\ list\ variable\rangle$ $\{\langle true\ code\rangle\}$ $\{\langle false\ code\rangle\}$

If the $\langle sequence\rangle$ 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 $\langle sequence\rangle$ is non-empty, stores the top 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.

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

`\seq_pop:cNTF`

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

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:NNTF <sequence> <token list variable> {(true code)} {(false code)}`

`\seq_gpop:cNTF`

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

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:Nn <sequence> {(item)}`

`\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)`

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.

13 Internal sequence functions

`\s__seq`
This scan mark (equal to `\scan_stop:`) marks the beginning of a sequence variable.

`__seq_item:n` * `__seq_item:n` $\{\langle item \rangle\}$
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:x`

`__seq_push_item_def:n` $\{\langle code \rangle\}$
Saves the definition of `__seq_item:n` and redefines it to accept one parameter and expand to $\langle code \rangle$. 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`.

Part IX

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 {⟨integer expression⟩}`

Evaluates the *⟨integer expression⟩*, expanding any integer and token list variables within the *⟨expression⟩* 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 *⟨integer expression⟩* may contain the operators `+`, `-`, `*` and `/`, along with parenthesis `(` and `)`. Any functions within the expressions should expand to an *⟨integer denotation⟩*: a sequence of a sign and digits matching the regex `\-?[0-9]+`. After expansion `\int_eval:n` yields an *⟨integer denotation⟩* which is left in the input stream.

TeXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an *⟨internal integer⟩*, and therefore requires suitable termination if used in a TeX-style integer assignment.

`\int_abs:n` ★ `\int_abs:n {⟨integer expression⟩}`

Updated: 2012-09-26

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

`\int_div_round:nn` ★ `\int_div_round:nn {⟨intexpr1⟩} {⟨intexpr2⟩}`

Updated: 2012-09-26

Evaluates the two *⟨integer expressions⟩* 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 `/` directly in an *⟨integer expression⟩*. The result is left in the input stream as an *⟨integer denotation⟩* after two expansions.

<code>\int_div_truncate:nn</code> *	<code>\int_div_truncate:nn {⟨integer expression⟩} {⟨integer expression⟩}</code>
Updated: 2012-02-09	Evaluates the two <i>⟨integer expressions⟩</i> 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 <i>⟨integer denotation⟩</i> after two expansions.

<code>\int_max:nn</code> *	<code>\int_max:nn {⟨integer expression⟩} {⟨integer expression⟩}</code>
<code>\int_min:nn</code> *	<code>\int_min:nn {⟨integer expression⟩} {⟨integer expression⟩}</code>
Updated: 2012-09-26	Evaluates the <i>⟨integer expressions⟩</i> as described for <code>\int_eval:n</code> and leaves either the larger or smaller value in the input stream as an <i>⟨integer denotation⟩</i> after two expansions.

<code>\int_mod:nn</code> *	<code>\int_mod:nn {⟨integer expression⟩} {⟨integer expression⟩}</code>
Updated: 2012-09-26	Evaluates the two <i>⟨integer expressions⟩</i> 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 {⟨integer expression⟩} {⟨integer expression⟩}</code> times <i>⟨integer expression⟩</i> from <i>⟨integer expression⟩</i> . Thus, the result has the same sign as <i>⟨integer expression⟩</i> and its absolute value is strictly less than that of <i>⟨integer expression⟩</i> . The result is left in the input stream as an <i>⟨integer denotation⟩</i> after two expansions.

2 Creating and initialising integers

<code>\int_new:N</code>	<code>\int_new:N ⟨integer⟩</code>
<code>\int_new:c</code>	Creates a new <i>⟨integer⟩</i> or raises an error if the name is already taken. The declaration is global. The <i>⟨integer⟩</i> is initially equal to 0.

<code>\int_const:Nn</code>	<code>\int_const:Nn ⟨integer⟩ {⟨integer expression⟩}</code>
<code>\int_const:cn</code>	Creates a new constant <i>⟨integer⟩</i> or raises an error if the name is already taken. The value of the <i>⟨integer⟩</i> is set globally to the <i>⟨integer expression⟩</i> .
Updated: 2011-10-22	

<code>\int_zero:N</code>	<code>\int_zero:N ⟨integer⟩</code>
<code>\int_zero:c</code>	Sets <i>⟨integer⟩</i> to 0.
<code>\int_gzero:N</code>	
<code>\int_gzero:c</code>	

<code>\int_zero_new:N</code>	<code>\int_zero_new:N ⟨integer⟩</code>
<code>\int_zero_new:c</code>	Ensures that the <i>⟨integer⟩</i> exists globally by applying <code>\int_new:N</code> if necessary, then applies <code>\int_(g)zero:N</code> to leave the <i>⟨integer⟩</i> set to zero.
<code>\int_gzero_new:N</code>	
<code>\int_gzero_new:c</code>	

New: 2011-12-13

<code>\int_set_eq:NN</code>	<code>\int_set_eq:NN ⟨integer₁⟩ ⟨integer₂⟩</code>
<code>\int_set_eq:(cN Nc cc)</code>	Sets the content of <i>⟨integer₁⟩</i> equal to that of <i>⟨integer₂⟩</i> .
<code>\int_gset_eq:NN</code>	
<code>\int_gset_eq:(cN Nc cc)</code>	

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

New: 2012-03-03

3 Setting and incrementing integers

<code>\int_add:Nn</code>	<code>\int_add:Nn</code> $\langle integer \rangle$ $\{\langle integer\ expression \rangle\}$
<code>\int_add:cn</code>	Adds the result of the $\langle integer\ expression \rangle$ to the current content of the $\langle integer \rangle$.
<code>\int_gadd:Nn</code>	
<code>\int_gadd:cn</code>	

Updated: 2011-10-22

<code>\int_decr:N</code>	<code>\int_decr:N</code> $\langle integer \rangle$
<code>\int_decr:c</code>	Decreases the value stored in $\langle integer \rangle$ by 1.
<code>\int_gdecr:N</code>	
<code>\int_gdecr:c</code>	

<code>\int_incr:N</code>	<code>\int_incr:N</code> $\langle integer \rangle$
<code>\int_incr:c</code>	Increases the value stored in $\langle integer \rangle$ by 1.
<code>\int_gincr:N</code>	
<code>\int_gincr:c</code>	

<code>\int_set:Nn</code>	<code>\int_set:Nn</code> $\langle integer \rangle$ $\{\langle integer\ expression \rangle\}$
<code>\int_set:cn</code>	Sets $\langle integer \rangle$ to the value of $\langle integer\ expression \rangle$, which must evaluate to an integer (as described for <code>\int_eval:n</code>).
<code>\int_gset:Nn</code>	
<code>\int_gset:cn</code>	

Updated: 2011-10-22

<code>\int_sub:Nn</code>	<code>\int_sub:Nn</code> $\langle integer \rangle$ $\{\langle integer\ expression \rangle\}$
<code>\int_sub:cn</code>	Subtracts the result of the $\langle integer\ expression \rangle$ from the current content of the $\langle integer \rangle$.
<code>\int_gsub:Nn</code>	
<code>\int_gsub:cn</code>	

Updated: 2011-10-22

4 Using integers

<code>\int_use:N</code> *	<code>\int_use:N</code> $\langle integer \rangle$
<code>\int_use:c</code> *	Recovers the content of an $\langle integer \rangle$ 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 an $\langle integer \rangle$ is required (such as in the first and third arguments of <code>\int_compare:nNnTF</code>).

Updated: 2011-10-22

TeXhackers note: `\int_use:N` is the TeX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

5 Integer expression conditionals

```

\int_compare_p:nNn * \int_compare_p:nNn {<intexpr1> <relation> {<intexpr2>}
\int_compare:nNnTF * \int_compare:nNnTF
  {<intexpr1> <relation> {<intexpr2>}
  {<true code>} {<false 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	!=

```

\int_case:nn  ☆ \int_case:nnTF {⟨test integer expression⟩}
\int_case:nnTF ☆ {
  {⟨intexpr case1⟩} {⟨code case1⟩}
  {⟨intexpr case2⟩} {⟨code case2⟩}
  ...
  {⟨intexpr casen⟩} {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false 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.

```

\int_if_even_p:n  ☆ \int_if_odd_p:n {⟨integer expression⟩}
\int_if_even:nTF ☆ \int_if_odd:nTF {⟨integer expression⟩}
\int_if_odd_p:n  ☆ {⟨true code⟩} {⟨false code⟩}
\int_if_odd:nTF ☆

```

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

```

\int_do_until:nNnn ☆ \int_do_until:nNnn {⟨intexpr1⟩} ⟨relation⟩ {⟨intexpr2⟩} {⟨code⟩}

```

Places the *⟨code⟩* in the input stream for \TeX 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**.

```

\int_do_while:nNnn ☆ \int_do_while:nNnn {⟨intexpr1⟩} ⟨relation⟩ {⟨intexpr2⟩} {⟨code⟩}

```

Places the *⟨code⟩* in the input stream for \TeX 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**.

`\int_until_do:nNnn` ☆ `\int_until_do:nNnn {<intexpr1> <relation> {<intexpr2>} {<code>}}`

Evaluates the relationship between the two *<integer expressions>* as described for `\int_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**.

`\int_while_do:nNnn` ☆ `\int_while_do:nNnn {<intexpr1> <relation> {<intexpr2>} {<code>}}`

Evaluates the relationship between the two *<integer expressions>* as described for `\int_compare:nNnTF`, and then places the *<code>* in the input stream if the *<relation>* is **true**. After the *<code>* has been processed by T_EX the test is repeated, and a loop occurs until the test is **false**.

`\int_do_until:nn` ☆ `\int_do_until:nn {<integer relation>} {<code>}}`

Updated: 2013-01-13

Places the *<code>* in the input stream for T_EX to process, and then evaluates the *<integer relation>* as described for `\int_compare:nTF`. If the test is **false** then the *<code>* is inserted into the input stream again and a loop occurs until the *<relation>* is **true**.

`\int_do_while:nn` ☆ `\int_do_while:nn {<integer relation>} {<code>}}`

Updated: 2013-01-13

Places the *<code>* in the input stream for T_EX to process, and then evaluates the *<integer relation>* as described for `\int_compare:nTF`. If the test is **true** then the *<code>* is inserted into the input stream again and a loop occurs until the *<relation>* is **false**.

`\int_until_do:nn` ☆ `\int_until_do:nn {<integer relation>} {<code>}}`

Updated: 2013-01-13

Evaluates the *<integer relation>* as described for `\int_compare:nTF`, 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**.

`\int_while_do:nn` ☆ `\int_while_do:nn {<integer relation>} {<code>}}`

Updated: 2013-01-13

Evaluates the *<integer relation>* as described for `\int_compare:nTF`, and then places the *<code>* in the input stream if the *<relation>* is **true**. After the *<code>* has been processed by T_EX the test is repeated, and a loop occurs until the test is **false**.

7 Integer step functions

`\int_step_function:nnnN` ☆

New: 2012-06-04

Updated: 2014-05-30

`\int_step_function:nnnN` {*initial value*} {*step*} {*final value*} {*function*}

This function first evaluates the *initial value*, *step* and *final value*, all of which should be integer expressions. 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 }
\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]
```

`\int_step_inline:nnnn`

New: 2012-06-04

Updated: 2014-05-30

`\int_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 integer expressions. 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`).

`\int_step_variable:nnnNn`

New: 2012-06-04

Updated: 2014-05-30

`\int_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 integer expressions. 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*.

8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

`\int_to_arabic:n` ☆

Updated: 2011-10-22

`\int_to_arabic:n` {*integer expression*}

Places the value of the *integer expression* in the input stream as digits, with category code 12 (other).

`\int_to_alpha:n` ★ `\int_to_alpha:n {⟨integer expression⟩}`

`\int_to_Alpha:n` ★

Updated: 2011-09-17

Evaluates the *⟨integer expression⟩* 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_alpha:n { 1 }
```

places a in the input stream,

```
\int_to_alpha:n { 26 }
```

is represented as z and

```
\int_to_alpha: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_alpha:n` and `\int_to_Alpha:n` functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

`\int_to_symbols:nnn` ★

Updated: 2011-09-17

```
\int_to_symbols:nnn
  {⟨integer expression⟩} {⟨total symbols⟩}
  {⟨value to symbol mapping⟩}
```

This is the low-level function for conversion of an *⟨integer expression⟩* into a symbolic form (often letters). The *⟨total symbols⟩* available should be given as an integer expression. Values are actually converted to symbols according to the *⟨value to symbol mapping⟩*. This should be given as *⟨total symbols⟩* pairs of entries, a number and the appropriate symbol. Thus the `\int_to_alpha:n` function is defined as

```
\cs_new:Npn \int_to_alpha:n #1
{
  \int_to_symbols:nnn {#1} { 26 }
  {
    { 1 } { a }
    { 2 } { b }
    ...
    { 26 } { z }
  }
}
```

`\int_to_bin:n` ★ `\int_to_bin:n {⟨integer expression⟩}`

New: 2014-02-11

Calculates the value of the *⟨integer expression⟩* and places the binary representation of the result in the input stream.

`\int_to_hex:n` ★ `\int_to_hex:n {⟨integer expression⟩}`

`\int_to_Hex:n` ★

New: 2014-02-11

Calculates the value of the *⟨integer expression⟩* 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 `\int_to_hex:n` and upper case ones for `\int_to_Hex:n`. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

`\int_to_oct:n` ★ `\int_to_oct:n {⟨integer expression⟩}`

New: 2014-02-11

Calculates the value of the *⟨integer expression⟩* 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).

`\int_to_base:nn` ★ `\int_to_base:nn {⟨integer expression⟩} {⟨base⟩}`

`\int_to_Base:nn` ★

Updated: 2014-02-11

Calculates the value of the *⟨integer expression⟩* and converts it into the appropriate representation in the *⟨base⟩*; 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 `\int_to_base:n` and upper case ones for `\int_to_Base:n`. The maximum *⟨base⟩* 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.*

`\int_to_roman:n` ☆ `\int_to_roman:n {⟨integer expression⟩}`

`\int_to_Roman:n` ☆

Updated: 2011-10-22

Places the value of the *⟨integer expression⟩* in the input stream as Roman numerals, either lower case (`\int_to_roman:n`) or upper case (`\int_to_Roman:n`). The Roman numerals are letters with category code 11 (letter).

9 Converting from other formats to integers

`\int_from_alph:n` ★ `\int_from_alph:n {⟨letters⟩}`

Updated: 2014-08-25

Converts the *⟨letters⟩* into the integer (base 10) representation and leaves this in the input stream. The *⟨letters⟩* 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 `\int_to_alph:n` and `\int_to_Alph:n`.

`\int_from_bin:n` ★ `\int_from_bin:n {⟨binary number⟩}`

New: 2014-02-11

Updated: 2014-08-25

Converts the *⟨binary number⟩* into the integer (base 10) representation and leaves this in the input stream. The *⟨binary number⟩* 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 `\int_to_bin:n`.

`\int_from_hex:n` ★ `\int_from_hex:n {⟨hexadecimal number⟩}`

New: 2014-02-11
Updated: 2014-08-25

Converts the *⟨hexadecimal number⟩* into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the *⟨hexadecimal number⟩* by upper or lower case letters. The *⟨hexadecimal number⟩* 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 `\int_to_hex:n` and `\int_to_Hex:n`.

`\int_from_oct:n` ★ `\int_from_oct:n {⟨octal number⟩}`

New: 2014-02-11
Updated: 2014-08-25

Converts the *⟨octal number⟩* into the integer (base 10) representation and leaves this in the input stream. The *⟨octal number⟩* 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 `\int_to_oct:n`.

`\int_from_roman:n` ★ `\int_from_roman:n {⟨roman numeral⟩}`

Updated: 2014-08-25

Converts the *⟨roman numeral⟩* into the integer (base 10) representation and leaves this in the input stream. The *⟨roman numeral⟩* is first converted to a string, with no expansion. The *⟨roman numeral⟩* 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 `\int_to_roman:n` and `\int_to_Roman:n`.

`\int_from_base:nn` ★ `\int_from_base:nn {⟨number⟩} {⟨base⟩}`

Updated: 2014-08-25

Converts the *⟨number⟩* expressed in *⟨base⟩* into the appropriate value in base 10. The *⟨number⟩* is first converted to a string, with no expansion. The *⟨number⟩* should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum *⟨base⟩* value is 36. This is the inverse function of `\int_to_base:nn` and `\int_to_Base:nn`.

10 Viewing integers

`\int_show:N` `\int_show:N ⟨integer⟩`
`\int_show:c`

Displays the value of the *⟨integer⟩* on the terminal.

`\int_show:n` `\int_show:n {⟨integer expression⟩}`

New: 2011-11-22
Updated: 2015-08-07

Displays the result of evaluating the *⟨integer expression⟩* on the terminal.

`\int_log:N` `\int_log:N ⟨integer⟩`
`\int_log:c`

Writes the value of the *⟨integer⟩* in the log file.

New: 2014-08-22
Updated: 2015-08-03

`\int_log:n` `\int_log:n {⟨integer expression⟩}`

New: 2014-08-22
Updated: 2015-08-07

Writes the result of evaluating the *⟨integer expression⟩* in the log file.

11 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.

12 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 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.

TeXhackers note: These are both names for the TeX primitive `\ifnum`.

`\if_case:w` ★ `\if_case:w` $\langle integer \rangle$ $\langle case_0 \rangle$
`\or:` ★ `\or:` $\langle case_1 \rangle$
 `\or:` ...
 `\else:` $\langle default \rangle$
`\fi:`

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`).

TeXhackers note: These are the TeX primitives `\ifcase` and `\or`.

`\if_int_odd:w` ★ `\if_int_odd:w` $\langle tokens \rangle$ $\langle optional\ space \rangle$
 $\langle true\ code \rangle$
`\else:`
 $\langle true\ code \rangle$
`\fi:`

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.

TeXhackers note: This is the TeX primitive `\ifodd`.

14 Internal functions

`__int_to_roman:w` ★ `__int_to_roman:w` $\langle integer \rangle$ $\langle space \rangle$ or $\langle non-expandable\ token \rangle$

Converts $\langle integer \rangle$ to its lower case Roman representation. Expansion ends when a space or non-expandable token is found. Note that this function produces a string of letters with category code 12 and that protected functions *are* expanded by this process. Negative $\langle integer \rangle$ values result in no output, although the function does not terminate expansion until a suitable endpoint is found in the same way as for positive numbers.

TeXhackers note: This is the TeX primitive `\romannumeral` renamed.

`__int_value:w` *

`__int_value:w` $\langle integer \rangle$
`__int_value:w` $\langle tokens \rangle$ $\langle optional\ space \rangle$

Expands $\langle tokens \rangle$ until an $\langle integer \rangle$ is formed. One space may be gobbled in the process.

TeXhackers note: This is the TeX primitive `\number`.

`__int_eval:w` *

`__int_eval:w` $\langle intexpr \rangle$ `__int_eval_end:`

`__int_eval_end:` *

Evaluates $\langle integer\ expression \rangle$ as described for `\int_eval:n`. The evaluation stops when an unexpandable token which is not a valid part of an integer is read or when `__int_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `__int_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

TeXhackers note: This is the ϵ -TeX primitive `\numexpr`.

`__prg_compare_error:`

`__prg_compare_error:`

`__prg_compare_error:Nw`

`__prg_compare_error:Nw` $\langle token \rangle$

These are used within `\int_compare:nTF`, `\dim_compare:nTF` and so on to recover correctly if the n-type argument does not contain a properly-formed relation.

Part X

The `l3intarray` package: low-level arrays of small integers

1 `l3intarray` documentation

This module provides no user function: at present it is meant for kernel use only.

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).

1.1 Internal functions

`__intarray_new:Nn``__intarray_new:Nn <intarray var> {<size>}`

Evaluates the integer expression `<size>` and allocates an `<integer array variable>` with that number of (zero) entries.

`__intarray_count:N *``__intarray_count:N <intarray var>`

Expands to the number of entries in the `<integer array variable>`. Contrarily to `\seq_count:N` this is performed in constant time.

`__intarray_gset:Nnn``__intarray_gset:Nnn <intarray var> {<position>} {<value>}``__intarray_gset_fast:Nnn``__intarray_gset_fast:Nnn <intarray var> {<position>} {<value>}`

Stores the result of evaluating the integer expression `<value>` into the `<integer array variable>` at the (integer expression) `<position>`. While `__intarray_gset:Nnn` checks that the `<position>` is between 1 and the `__intarray_count:N` and that the `<value>`’s absolute value is at most $2^{30} - 1$, the “fast” function performs no such bound check. Assignments are always global.

`__intarray_item:Nn``__intarray_item:Nn <intarray var> {<position>}``__intarray_item_fast:Nn *``__intarray_item_fast:Nn <intarray var> {<position>}`

Expands to the integer entry stored at the (integer expression) `<position>` in the `<integer array variable>`. While `__intarray_item:Nn` checks that the `<position>` is between 1 and the `__intarray_count:N`, the “fast” function performs no such bound check.

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

`\flag_new:n` `\flag_new:n {<flag name>}`

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.

`\flag_clear:n` `\flag_clear:n {<flag name>}`

The *flag*'s height is set to zero. The assignment is local.

`\flag_clear_new:n` `\flag_clear_new:n {<flag name>}`

Ensures that the *flag* exists globally by applying `\flag_new:n` if necessary, then applies `\flag_clear:n`, setting the height to zero locally.

`\flag_show:n` `\flag_show:n {<flag name>}`

Displays the *flag*'s height in the terminal.

`\flag_log:n` `\flag_log:n {<flag name>}`

Writes the *flag*'s height to the log file.

2 Expandable flag commands

`\flag_if_exist_p:n *` `\flag_if_exist:n {⟨flag name⟩}`

`\flag_if_exist:n \underline{TF} *` This function returns `true` if the `⟨flag name⟩` references a flag that has been defined previously, and `false` otherwise.

`\flag_if_raised_p:n *` `\flag_if_raised:n {⟨flag name⟩}`

`\flag_if_raised:n \underline{TF} *` This function returns `true` if the `⟨flag⟩` has non-zero height, and `false` if the `⟨flag⟩` has zero height.

`\flag_height:n *` `\flag_height:n {⟨flag name⟩}`

Expands to the height of the `⟨flag⟩` as an integer denotation.

`\flag_raise:n *` `\flag_raise:n {⟨flag name⟩}`

The `⟨flag⟩`'s height is increased by 1 locally.

Part XII

The l3quark package

Quarks

1 Introduction to quarks and scan marks

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_`. Scan marks are for internal use by the kernel: they are not intended for more general use.

1.1 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}
```

`\q_mark` Used as a marker for delimited arguments when `\q_stop` is already in use.

`\q_nil` 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 `\q_stop`, which is only ever used as a delimiter).

`\q_no_value` 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 `\prop_get:NnN` 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.

`\quark_if_nil_p:N` * `\quark_if_nil_p:N` $\langle token \rangle$
`\quark_if_nil:NTF` * `\quark_if_nil:NTF` $\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle token \rangle$ is equal to `\q_nil`.

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

`\quark_if_no_value_p:N` * `\quark_if_no_value_p:N` $\langle token \rangle$
`\quark_if_no_value_p:c` * `\quark_if_no_value:NTF` $\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
`\quark_if_no_value:NTF` * Tests if the $\langle token \rangle$ is equal to `\q_no_value`.
`\quark_if_no_value:cTF` *

`\quark_if_no_value_p:n` * `\quark_if_no_value_p:n` $\{\langle token list \rangle\}$
`\quark_if_no_value:nTF` * `\quark_if_no_value:nTF` $\{\langle token list \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle token list \rangle$ contains only `\q_no_value` (distinct from $\langle token list \rangle$ 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.

`\q_recursion_tail` 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.

`\q_recursion_stop` This quark is added *after* 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` $\langle token \rangle$

Tests if $\langle token \rangle$ 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` $\{\langle token list \rangle\}$
`\quark_if_recursion_tail_stop:o`

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ 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` $\langle token \rangle$ $\{\langle insertion \rangle\}$

Tests if $\langle token \rangle$ 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 $\langle insertion \rangle$ 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` $\{\langle token list \rangle\}$ $\{\langle insertion \rangle\}$
`\quark_if_recursion_tail_stop_do:on`

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ 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 $\langle insertion \rangle$ code is then added to the input stream after the recursion has ended.

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_db1:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_db1: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 Internal quark functions

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

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:.`

7 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`).

The scan marks system is only for internal use by the kernel team in a small number of very specific places. These functions should not be used more generally.

`_scan_new:N` `_scan_new:N` $\langle scan\ mark \rangle$
Creates a new $\langle scan\ mark \rangle$ which is set equal to `\scan_stop:.` The $\langle scan\ mark \rangle$ is defined globally, and an error message is raised if the name was already taken by another scan mark.

`\s__stop` Used at the end of a set of instructions, as a marker that can be jumped to using `_use_none_delimit_by_s__stop:w`.

`_use_none_delimit_by_s__stop:w` `_use_none_delimit_by_s__stop:w` $\langle tokens \rangle$ `\s__stop`
Removes the $\langle tokens \rangle$ and `\s__stop` from the input stream. This leads to a low-level TeX error if `\s__stop` is absent.

Part XIII

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.

```
\prg_new_eq_conditional:NNn \prg_new_eq_conditional:NNn \<name1>:<arg spec1> \<name2>:<arg spec2>
\prg_set_eq_conditional:NNn {\<conditions>}
```

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`.

```

\prg_return_true:  * \prg_return_true:
\prg_return_false: * \prg_return_false:

```

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).

TeXhackers note: The `bool` data type is not implemented using the `\iffalse/\iftrue` primitives, in contrast to `\newif`, *etc.*, in plain TeX, L^ATeX 2_ε and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

```

\bool_new:N \bool_new:N <boolean>
\bool_new:c

```

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

```

\bool_set_false:N \bool_set_false:N <boolean>
\bool_set_false:c
\bool_gset_false:N
\bool_gset_false:c

```

Sets `<boolean>` logically `false`.

```

\bool_set_true:N \bool_set_true:N <boolean>
\bool_set_true:c
\bool_gset_true:N
\bool_gset_true:c

```

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` $\langle boolean \rangle$ $\{\langle boolexpr \rangle\}$
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` $\langle boolean \rangle$ $\{\langle true \text{ code} \rangle\}$ $\{\langle false \text{ code} \rangle\}$
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` $\langle boolean \rangle$
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` $\langle boolean \rangle$
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` $\langle boolean \rangle$ $\{\langle true \text{ code} \rangle\}$ $\{\langle false \text{ code} \rangle\}$
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 *⟨true⟩* or *⟨false⟩* values, it seems only fitting that we also provide a parser for *⟨boolean expressions⟩*.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean *⟨true⟩* or *⟨false⟩*. 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`.

```
\bool_if_p:n * \bool_if_p:n {<boolean expression>}
\bool_if:nTF * \bool_if:nTF {<boolean expression>} {<true code>} {<false code>}
```

Updated: 2017-07-15 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.

```
\bool_lazy_all_p:n * \bool_lazy_all_p:n { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} }
\bool_lazy_all:nTF * \bool_lazy_all:nTF { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} } {<true code>}
{<false code>}
```

New: 2015-11-15
Updated: 2017-07-15 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>*.

```
\bool_lazy_and_p:nn * \bool_lazy_and_p:nn {<boolexpr1>} {<boolexpr2>}
\bool_lazy_and:nnTF * \bool_lazy_and:nnTF {<boolexpr1>} {<boolexpr2>} {<true code>} {<false code>}
```

New: 2015-11-15
Updated: 2017-07-15 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>*.

```
\bool_lazy_any_p:n * \bool_lazy_any_p:n { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} }
\bool_lazy_any:nTF * \bool_lazy_any:nTF { {<boolexpr1>} {<boolexpr2>} ... {<boolexprN>} } {<true code>}
{<false code>}
```

New: 2015-11-15
Updated: 2017-07-15 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>*.

```
\bool_lazy_or_p:nn * \bool_lazy_or_p:nn {<boolexpr1>} {<boolexpr2>}
\bool_lazy_or:nnTF * \bool_lazy_or:nnTF {<boolexpr1>} {<boolexpr2>} {<true code>} {<false code>}
```

New: 2015-11-15
Updated: 2017-07-15 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>*.

```
\bool_not_p:n * \bool_not_p:n {<boolean expression>}
```

Updated: 2017-07-15 Function version of `!(<boolean expression>)` within a boolean expression.

`\bool_xor_p:nm` ☆ `\bool_xor_p:nm {<boolexpr1>} {<boolexpr2>}`

Updated: 2017-07-15

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.

`\bool_do_until:Nn` ☆ `\bool_do_until:Nn <boolean> {<code>}`

`\bool_do_until:cn` ☆

Updated: 2017-07-15

Places the `<code>` in the input stream for \TeX to process, and then checks the logical value of the `<boolean>`. If it is `false` then the `<code>` is inserted into the input stream again and the process loops until the `<boolean>` is `true`.

`\bool_do_while:Nn` ☆ `\bool_do_while:Nn <boolean> {<code>}`

`\bool_do_while:cn` ☆

Updated: 2017-07-15

Places the `<code>` in the input stream for \TeX to process, and then checks the logical value of the `<boolean>`. If it is `true` then the `<code>` is inserted into the input stream again and the process loops until the `<boolean>` is `false`.

`\bool_until_do:Nn` ☆ `\bool_until_do:Nn <boolean> {<code>}`

`\bool_until_do:cn` ☆

Updated: 2017-07-15

This function firsts checks the logical value of the `<boolean>`. If it is `false` the `<code>` is placed in the input stream and expanded. After the completion of the `<code>` the truth of the `<boolean>` is re-evaluated. The process then loops until the `<boolean>` is `true`.

`\bool_while_do:Nn` ☆ `\bool_while_do:Nn <boolean> {<code>}`

`\bool_while_do:cn` ☆

Updated: 2017-07-15

This function firsts checks the logical value of the `<boolean>`. If it is `true` the `<code>` is placed in the input stream and expanded. After the completion of the `<code>` the truth of the `<boolean>` is re-evaluated. The process then loops until the `<boolean>` is `false`.

`\bool_do_until:nm` ☆ `\bool_do_until:nm {<boolean expression>} {<code>}`

Updated: 2017-07-15

Places the `<code>` in the input stream for \TeX to process, and then checks the logical value of the `<boolean expression>` as described for `\bool_if:nTF`. If it is `false` then the `<code>` is inserted into the input stream again and the process loops until the `<boolean expression>` evaluates to `true`.

`\bool_do_while:nm` ☆ `\bool_do_while:nm {<boolean expression>} {<code>}`

Updated: 2017-07-15

Places the `<code>` in the input stream for \TeX to process, and then checks the logical value of the `<boolean expression>` as described for `\bool_if:nTF`. If it is `true` then the `<code>` is inserted into the input stream again and the process loops until the `<boolean expression>` evaluates to `false`.

`\bool_until_do:nm` ☆ `\bool_until_do:nm {<boolean expression>} {<code>}`

Updated: 2017-07-15

This function firsts checks the logical value of the `<boolean expression>` (as described for `\bool_if:nTF`). If it is `false` the `<code>` is placed in the input stream and expanded. After the completion of the `<code>` the truth of the `<boolean expression>` is re-evaluated. The process then loops until the `<boolean expression>` 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 Internal programming functions

`\group_align_safe_begin:` ★
`\group_align_safe_end:` ★

Updated: 2011-08-11

`\group_align_safe_begin:`
...
`\group_align_safe_end:`

These functions are used to enclose material in a T_EX 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 T_EX 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.

`__prg_break_point:Nn` ★

`__prg_break_point:Nn \<type>_map_break: {<tokens>}`

Used to mark the end of a recursion or mapping: the functions `\<type>_map_break:` and `\<type>_map_break:n` use this to break out of the loop. After the loop ends, the *<tokens>* are inserted into the input stream. This occurs even if the break functions are *not* applied: `__prg_break_point:Nn` is functionally-equivalent in these cases to `\use_ii:nn`.

`__prg_map_break:Nn` ★

`__prg_map_break:Nn \<type>_map_break: {<user code>}`

...

`__prg_break_point:Nn \<type>_map_break: {<ending code>}`

Breaks a recursion in mapping contexts, inserting in the input stream the *<user code>* after the *<ending code>* for the loop. The function breaks loops, inserting their *<ending code>*, until reaching a loop with the same *<type>* as its first argument. This `\<type>_map_break:` argument is simply used as a recognizable marker for the *<type>*.

`\g__prg_map_int`

This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions `__prg_map_1:w`, `__prg_map_2:w`, *etc.*, labelled by `\g__prg_map_int` hold functions to be mapped over various list datatypes in inline and variable mappings.

`__prg_break_point:` ★

This copy of `\prg_do_nothing:` is used to mark the end of a fast short-term recursion: the function `__prg_break:n` uses this to break out of the loop.

`__prg_break:` ★

`__prg_break:n {<tokens>} ... __prg_break_point:`

`__prg_break:n` ★

Breaks a recursion which has no *<ending code>* and which is not a user-breakable mapping (see for instance `\prop_get:Nn`), and inserts *<tokens>* in the input stream.

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 resulting ordered list can then be mapped over using `\clist_map_function:NN`. Several items can be added at once, and spaces are removed from both sides of each item on input. Hence,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~ a ~ , ~ {b} ~ }
\clist_put_right:Nn \l_my_clist { ~ { c ~ } , d }
```

results in `\l_my_clist` containing `a,{b},{c~},d`. Comma lists cannot contain empty items, thus

```
\clist_clear_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

leaves `true` in the input stream. To include an item which contains a comma, or starts or ends with a space, surround it with braces. The sequence data type should be preferred to comma lists if items are to contain `{`, `}`, or `#` (assuming the usual \TeX category codes apply).

1 Creating and initialising comma lists

```
\clist_new:N
\clist_new:c
```

```
\clist_new:N <comma list>
```

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.

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

```
\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>*.

New: 2014-07-05

```
\clist_clear:N
\clist_clear:c
\clist_gclear:N
\clist_gclear:c
```

```
\clist_clear:N <comma list>
```

Clears all items from the *<comma list>*.

```
\clist_clear_new:N
\clist_clear_new:c
\clist_gclear_new:N
\clist_gclear_new:c
```

```
\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.

<code>\clist_set_eq:NN</code>	<code>\clist_set_eq:NN</code> $\langle comma list_1 \rangle$ $\langle comma list_2 \rangle$
<code>\clist_set_eq:(cN Nc cc)</code>	Sets the content of $\langle comma list_1 \rangle$ equal to that of $\langle comma list_2 \rangle$.
<code>\clist_gset_eq:NN</code>	
<code>\clist_gset_eq:(cN Nc cc)</code>	

<code>\clist_set_from_seq:NN</code>	<code>\clist_set_from_seq:NN</code> $\langle comma list \rangle$ $\langle sequence \rangle$
<code>\clist_set_from_seq:(cN Nc cc)</code>	
<code>\clist_gset_from_seq:NN</code>	
<code>\clist_gset_from_seq:(cN Nc cc)</code>	

New: 2014-07-17

Converts the data in the $\langle sequence \rangle$ into a $\langle comma list \rangle$: the original $\langle sequence \rangle$ is unchanged. Items which contain either spaces or commas are surrounded by braces.

<code>\clist_concat:NNN</code>	<code>\clist_concat:NNN</code> $\langle comma list_1 \rangle$ $\langle comma list_2 \rangle$ $\langle comma list_3 \rangle$
<code>\clist_concat:ccc</code>	Concatenates the content of $\langle comma list_2 \rangle$ and $\langle comma list_3 \rangle$ together and saves the result in $\langle comma list_1 \rangle$. The items in $\langle comma list_2 \rangle$ are placed at the left side of the new comma list.
<code>\clist_gconcat:NNN</code>	
<code>\clist_gconcat:ccc</code>	

<code>\clist_if_exist_p:N *</code>	<code>\clist_if_exist_p:N</code> $\langle comma list \rangle$
<code>\clist_if_exist_p:c *</code>	<code>\clist_if_exist:NTF</code> $\langle comma list \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$
<code>\clist_if_exist:NTF *</code>	Tests whether the $\langle comma list \rangle$ is currently defined. This does not check that the $\langle comma list \rangle$ really is a comma list.
<code>\clist_if_exist:cTF *</code>	

New: 2012-03-03

2 Adding data to comma lists

<code>\clist_set:Nn</code>	<code>\clist_set:Nn</code> $\langle comma list \rangle$ $\{\langle item_1 \rangle, \dots, \langle item_n \rangle\}$
<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. Spaces are removed from both sides of each item.

<code>\clist_put_left:Nn</code>	<code>\clist_put_left:Nn</code> $\langle comma list \rangle$ $\{\langle item_1 \rangle, \dots, \langle item_n \rangle\}$
<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 $\langle comma list \rangle$. Spaces are removed from both sides of each item.

```

\clist_put_right:Nn          \clist_put_right:Nn <comma list> {<item_1>,...,<item_n>}
\clist_put_right:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_right:Nn
\clist_gput_right:(NV|No|Nx|cn|cV|co|cx)

```

Updated: 2011-09-05

Appends the *<items>* to the right of the *<comma list>*. Spaces are removed from both sides of each item.

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.

```

\clist_remove_duplicates:N   \clist_remove_duplicates:N <comma list>
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c

```

Removes duplicate items from the *<comma list>*, leaving the left most copy of each item in the *<comma list>*. The *<item>* comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

TeXhackers note: This function iterates through every item in the *<comma list>* and does a comparison with the *<items>* already checked. It is therefore relatively slow with large comma lists. Furthermore, it does not work if any of the items in the *<comma list>* contains `{`, `}`, or `#` (assuming the usual TeX category codes apply).

```

\clist_remove_all:Nn        \clist_remove_all:Nn <comma list> {<item>}
\clist_remove_all:cn
\clist_gremove_all:Nn
\clist_gremove_all:cn

```

Removes every occurrence of *<item>* from the *<comma list>*. The *<item>* comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

Updated: 2011-09-06

TeXhackers note: The *<item>* may not contain `{`, `}`, or `#` (assuming the usual TeX category codes apply).

```

\clist_reverse:N           \clist_reverse:N <comma list>
\clist_reverse:c
\clist_greverse:N
\clist_greverse:c

```

New: 2014-07-18

```

\clist_reverse:n          \clist_reverse:n {<comma list>}

```

Leaves the items in the *<comma list>* in the input stream in reverse order. Braces and spaces are preserved by this process.

New: 2014-07-18

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>	Sorts the items in the <i><clist var></i> according to the <i><comparison code></i> , and assigns the result to <i><clist var></i> . The details of sorting comparison are described in Section 1.
<code>\clist_gsort:cn</code>	

New: 2017-02-06

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:NTF <comma list> {<true code>} {<false code>}</code>
<code>\clist_if_empty:NTF</code> *	Tests if the <i><comma list></i> is empty (containing no items).
<code>\clist_if_empty:cTF</code> *	

<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 *<comma list>* is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list `{~,~,~}` (without outer braces) is empty, while `{~, { }, }` (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 *<item>* is present in the *<comma list>*. In the case of an n-type *<comma list>*, spaces are stripped from each item, but braces are not removed. Hence,

```
\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}
```

yields false.

TeXhackers note: The *<item>* may not contain `{`, `}`, or `#` (assuming the usual TeX category codes apply), and should not contain `,` nor start or end with a space.

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 your comma list that is being mapped is `{a_ ,_{b}_ ,_{ },_{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.

`\clist_map_function:NN` ☆
`\clist_map_function:cN` ☆
`\clist_map_function:nN` ☆

Updated: 2012-06-29

`\clist_map_function:NN` *<comma list>* *<function>*

Applies *<function>* to every *<item>* stored in the *<comma list>*. The *<function>* receives one argument for each iteration. The *<items>* are returned from left to right. The function `\clist_map_inline:Nn` is in general more efficient than `\clist_map_function:NN`. One mapping may be nested inside another.

`\clist_map_inline:Nn`
`\clist_map_inline:cN`
`\clist_map_inline:nN`

Updated: 2012-06-29

`\clist_map_inline:Nn` *<comma list>* *{<inline function>}*

Applies *<inline function>* to every *<item>* stored within the *<comma list>*. The *<inline function>* should consist of code which receives the *<item>* as #1. One in line mapping can be nested inside another. The *<items>* are returned from left to right.

`\clist_map_variable:NNn`
`\clist_map_variable:cNn`
`\clist_map_variable:nNn`

Updated: 2012-06-29

`\clist_map_variable:NNn` *<comma list>* *<tl var.>* *{<function using tl var.>}*

Stores each entry in the *<comma list>* in turn in the *<tl var.>* and applies the *<function using tl var.>* The *<function>* usually consists of code making use of the *<tl var.>*, but this is not enforced. One variable mapping can be nested inside another. The *<items>* are returned from left to right.

`\clist_map_break:` ☆

Updated: 2012-06-29

`\clist_map_break:`

Used to terminate a `\clist_map_...` function before all entries in the *<comma list>* 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 T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before further items are taken from the input stream. This depends on the design of the mapping function.

`\clist_map_break:n` ☆

Updated: 2012-06-29

`\clist_map_break:n` {<tokens>}

Used to terminate a `\clist_map_...` function before all entries in the <comma list> have been processed, inserting the <tokens> 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 { <tokens> } }
  {
    % Do something useful
  }
}
```

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

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before the <tokens> are inserted into the input stream. This depends on the design of the mapping function.

`\clist_count:N` ☆

`\clist_count:c` ☆

`\clist_count:n` ☆

New: 2012-07-13

`\clist_count:N` <comma list>

Leaves the number of items in the <comma list> in the input stream as an <integer denotation>. The total number of items in a <comma list> includes those which are duplicates, *i.e.* every item in a <comma list> is unique.

6 Using the content of comma lists directly

`\clist_use:Nnnn` ☆

`\clist_use:cnnn` ☆

New: 2013-05-26

`\clist_use:Nnnn` <clist var> {<separator between two>}

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

Places the contents of the <clist var> in the input stream, with the appropriate <separator> between the items. Namely, if the comma list has more than two items, the <separator between more than two> is placed between each pair of items except the last, for which the <separator between final two> is used. If the comma list has exactly two items, then they are placed in the input stream separated by the <separator between two>. 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 <items> do not expand further when appearing in an x-type argument expansion.

`\clist_use:Nn` *★* `\clist_use:Nn <clist var> {<separator>}`

`\clist_use:cn` *★*

New: 2013-05-26

Places the contents of the *<clist var>* in the input stream, with the *<separator>* 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.

T_EXhackers 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.

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.

`\clist_get:NN` `\clist_get:NN <comma list> <token list variable>`

`\clist_get:cN`

Updated: 2012-05-14

Stores the left-most item from a *<comma list>* in the *<token list variable>* without removing it from the *<comma list>*. The *<token list variable>* is assigned locally. If the *<comma list>* is empty the *<token list variable>* is set to the marker value `\q_no_value`.

`\clist_get:NNTF` `\clist_get:NNTF <comma list> <token list variable> {<true code>} {<false code>}`

`\clist_get:cNNTF`

New: 2012-05-14

If the *<comma list>* 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 *<comma list>* is non-empty, stores the top item from the *<comma list>* in the *<token list variable>* without removing it from the *<comma list>*. The *<token list variable>* is assigned locally.

`\clist_pop:NN` `\clist_pop:NN <comma list> <token list variable>`

`\clist_pop:cN`

Updated: 2011-09-06

Pops the left-most item from a *<comma list>* into the *<token list variable>*, *i.e.* removes the item from the comma list and stores it in the *<token list variable>*. Both of the variables are assigned locally.

`\clist_gpop:NN` `\clist_gpop:NN <comma list> <token list variable>`

`\clist_gpop:cN`

Pops the left-most item from a *<comma list>* into the *<token list variable>*, *i.e.* removes the item from the comma list and stores it in the *<token list variable>*. The *<comma list>* is modified globally, while the assignment of the *<token list variable>* is local.

`\clist_pop:NNTF` `\clist_pop:NNTF <comma list> <token list variable> {(true code)} {(false code)}`

`\clist_pop:cNTF`

New: 2012-05-14

If the `<comma list>` 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 `<comma list>` is non-empty, pops the top item from the `<comma list>` in the `<token list variable>`, *i.e.* removes the item from the `<comma list>`. Both the `<comma list>` and the `<token list variable>` are assigned locally.

`\clist_gpop:NNTF` `\clist_gpop:NNTF <comma list> <token list variable> {(true code)} {(false code)}`

`\clist_gpop:cNTF`

New: 2012-05-14

If the `<comma list>` 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 `<comma list>` is non-empty, pops the top item from the `<comma list>` in the `<token list variable>`, *i.e.* removes the item from the `<comma list>`. The `<comma list>` is modified globally, while the `<token list variable>` is assigned locally.

`\clist_push:Nn` `\clist_push:Nn <comma list> {(items)}`

`\clist_push:(NV|No|Nx|cn|cV|co|cx)`

`\clist_gpush:Nn`

`\clist_gpush:(NV|No|Nx|cn|cV|co|cx)`

Adds the `{(items)}` to the top of the `<comma list>`. Spaces are removed from both sides of each item.

8 Using a single item

`\clist_item:Nn *` `\clist_item:Nn <comma list> {(integer expression)}`

`\clist_item:cn *`

`\clist_item:nn *`

New: 2014-07-17

Indexing items in the `<comma list>` from 1 at the top (left), this function evaluates the `<integer expression>` and leaves the appropriate item from the comma list in the input stream. If the `<integer expression>` is negative, indexing occurs from the bottom (right) of the comma list. When the `<integer expression>` is larger than the number of items in the `<comma list>` (as calculated by `\clist_count:N`) then the function expands to nothing.

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

`\clist_show:N` `\clist_show:N <comma list>`

`\clist_show:c`

Updated: 2015-08-03

Displays the entries in the `<comma list>` in the terminal.

`\clist_show:n` `\clist_show:n {(tokens)}`

Updated: 2013-08-03

Displays the entries in the comma list in the terminal.

`\clist_log:N` `\clist_log:N` *<comma list>*
`\clist_log:c`
New: 2014-08-22
Updated: 2015-08-03

Writes the entries in the *<comma list>* in the log file. See also `\clist_show:N` which displays the result in the terminal.

`\clist_log:n` `\clist_log:n` *{<tokens>}*

New: 2014-08-22

Writes the entries in the comma list in the log file. See also `\clist_show:n` which displays the result in the terminal.

10 Constant and scratch comma lists

`\c_empty_clist`

New: 2012-07-02

Constant that is always empty.

`\l_tmpa_clist`
`\l_tmpb_clist`

New: 2011-09-06

Scratch comma lists 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_clist`
`\g_tmpb_clist`

New: 2011-09-06

Scratch comma lists 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.

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.

`\char_generate:nm` ★

New: 2015-09-09

`\char_generate:nm` { $\langle charcode \rangle$ } { $\langle catcode \rangle$ }

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

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 11 (letter)
- 12 (other)

and other values raise an error.

The $\langle charcode \rangle$ may be any one valid for the engine in use. Note however that for X_YTeX releases prior to 0.99992 only the 8-bit range (0 to 255) is accepted due to engine limitations.

`\c_catcode_other_space_tl`

New: 2011-09-05

Token list containing one character with category code 12, (“other”), and character code 32 (space).

2 Manipulating and interrogating character tokens

```
\char_set_catcode_escape:N          \char_set_catcode_letter:N <character>
\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
```

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.

```
\char_set_catcode_escape:n          \char_set_catcode_letter:n {\integer expression}
\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
```

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.

`\char_set_catcode:nn` `\char_set_catcode:nn {⟨integer1⟩} {⟨integer2⟩}`
Updated: 2015-11-11

These functions set the category code of the *⟨character⟩* which has character code as given by the *⟨integer expression⟩*. The first *⟨integer expression⟩* is the character code and the second is the category code to apply. The setting applies within the current T_EX group. In general, the symbolic functions `\char_set_catcode_⟨type⟩` should be preferred, but there are cases where these lower-level functions may be useful.

`\char_value_catcode:n` ★ `\char_value_catcode:n {⟨integer expression⟩}`

Expands to the current category code of the *⟨character⟩* with character code given by the *⟨integer expression⟩*.

`\char_show_value_catcode:n` `\char_show_value_catcode:n {⟨integer expression⟩}`

Displays the current category code of the *⟨character⟩* with character code given by the *⟨integer expression⟩* on the terminal.

`\char_set_lccode:nn` `\char_set_lccode:nn {⟨integer1⟩} {⟨integer2⟩}`
Updated: 2015-08-06

Sets up the behaviour of the *⟨character⟩* when found inside `\tl_lower_case:n`, such that *⟨character₁⟩* will be converted into *⟨character₂⟩*. The two *⟨characters⟩* may be specified using an *⟨integer expression⟩* for the character code concerned. This may include the T_EX ‘*⟨character⟩*’ method for converting a single character into its character code:

```
\char_set_lccode:nn { '\A } { '\a } % Standard behaviour
\char_set_lccode:nn { '\A } { '\A + 32 }
\char_set_lccode:nn { 50 } { 60 }
```

The setting applies within the current T_EX group.

`\char_value_lccode:n` ★ `\char_value_lccode:n {⟨integer expression⟩}`

Expands to the current lower case code of the *⟨character⟩* with character code given by the *⟨integer expression⟩*.

`\char_show_value_lccode:n` `\char_show_value_lccode:n {⟨integer expression⟩}`

Displays the current lower case code of the *⟨character⟩* with character code given by the *⟨integer expression⟩* on the terminal.

`\char_set_uccode:nn` `\char_set_uccode:nn {⟨integer1⟩} {⟨integer2⟩}`
Updated: 2015-08-06

Sets up the behaviour of the *⟨character⟩* when found inside `\tl_upper_case:n`, such that *⟨character₁⟩* will be converted into *⟨character₂⟩*. The two *⟨characters⟩* may be specified using an *⟨integer expression⟩* for the character code concerned. This may include the T_EX ‘*⟨character⟩*’ method for converting a single character into its character code:

```
\char_set_uccode:nn { '\a } { '\A } % Standard behaviour
\char_set_uccode:nn { '\A } { '\A - 32 }
\char_set_uccode:nn { 60 } { 50 }
```

The setting applies within the current T_EX group.

<code>\char_value_uccode:n</code> *	<code>\char_value_uccode:n</code> $\langle integer\ expression \rangle$
	Expands to the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer\ expression \rangle$.
<code>\char_show_value_uccode:n</code>	<code>\char_show_value_uccode:n</code> $\langle integer\ expression \rangle$
	Displays the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer\ expression \rangle$ on the terminal.
<code>\char_set_mathcode:nn</code> <small>Updated: 2015-08-06</small>	<code>\char_set_mathcode:nn</code> $\langle intexpr_1 \rangle$ $\langle intexpr_2 \rangle$
	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.
<code>\char_value_mathcode:n</code> *	<code>\char_value_mathcode:n</code> $\langle integer\ expression \rangle$
	Expands to the current math code of the $\langle character \rangle$ with character code given by the $\langle integer\ expression \rangle$.
<code>\char_show_value_mathcode:n</code>	<code>\char_show_value_mathcode:n</code> $\langle integer\ expression \rangle$
	Displays the current math code of the $\langle character \rangle$ with character code given by the $\langle integer\ expression \rangle$ on the terminal.
<code>\char_set_sfcode:nn</code> <small>Updated: 2015-08-06</small>	<code>\char_set_sfcode:nn</code> $\langle intexpr_1 \rangle$ $\langle intexpr_2 \rangle$
	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.
<code>\char_value_sfcode:n</code> *	<code>\char_value_sfcode:n</code> $\langle integer\ expression \rangle$
	Expands to the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer\ expression \rangle$.
<code>\char_show_value_sfcode:n</code>	<code>\char_show_value_sfcode:n</code> $\langle integer\ expression \rangle$
	Displays the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer\ expression \rangle$ on the terminal.
<code>\l_char_active_seq</code> <small>New: 2012-01-23 Updated: 2015-11-11</small>	
	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 \backslash . Active tokens should be added to the sequence when they are defined for general document use.
<code>\l_char_special_seq</code> <small>New: 2012-01-23 Updated: 2015-11-11</small>	
	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 \backslash for the backslash or $\{$ for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.

3 Generic tokens

`\token_new:Nn` `\token_new:Nn <token1> {<token2>}`

Defines `<token1>` to globally be a snapshot of `<token2>`. This is an implicit representation of `<token2>`.

`\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:N <token>`
`\token_to_meaning:c` ★

Inserts the current meaning of the `<token>` into the input stream as a series of characters of category code 12 (other). This is the primitive `TEX` description of the `<token>`, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` are described as macros.

T_EXhackers note: This is the `TEX` primitive `\meaning`.

`\token_to_str:N` ★ `\token_to_str:N <token>`
`\token_to_str:c` ★

Converts the given `<token>` into a series of characters with category code 12 (other). If the `<token>` is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the `<token>`). This function requires only a single expansion.

T_EXhackers note: `\token_to_str:N` is the `TEX` primitive `\string` renamed.

5 Token conditionals

```
\token_if_group_begin_p:N * \token_if_group_begin_p:N <token>
\token_if_group_begin:NTF * \token_if_group_begin:NTF <token> {\true code} {\false code}
```

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.

```
\token_if_group_end_p:N * \token_if_group_end_p:N <token>
\token_if_group_end:NTF * \token_if_group_end:NTF <token> {\true code} {\false code}
```

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.

```
\token_if_math_toggle_p:N * \token_if_math_toggle_p:N <token>
\token_if_math_toggle:NTF * \token_if_math_toggle:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of a math shift token ($\$$ when normal \TeX category codes are in force).

```
\token_if_alignment_p:N * \token_if_alignment_p:N <token>
\token_if_alignment:NTF * \token_if_alignment:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of an alignment token ($\&$ when normal \TeX category codes are in force).

```
\token_if_parameter_p:N * \token_if_parameter_p:N <token>
\token_if_parameter:NTF * \token_if_parameter:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of a macro parameter token ($\#$ when normal \TeX category codes are in force).

```
\token_if_math_superscript_p:N * \token_if_math_superscript_p:N <token>
\token_if_math_superscript:NTF * \token_if_math_superscript:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of a superscript token ($\^$ when normal \TeX category codes are in force).

```
\token_if_math_subscript_p:N * \token_if_math_subscript_p:N <token>
\token_if_math_subscript:NTF * \token_if_math_subscript:NTF <token> {\true code} {\false code}
```

Tests if $\langle token \rangle$ has the category code of a subscript token ($_$ when normal \TeX category codes are in force).

```
\token_if_space_p:N * \token_if_space_p:N <token>
\token_if_space:NTF * \token_if_space:NTF <token> {\true code} {\false code}
```

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.

```
\token_if_letter_p:N * \token_if_letter_p:N <token>
\token_if_letter:NTF * \token_if_letter:NTF <token> {(true code)} {(false code)}
```

Tests if $\langle token \rangle$ has the category code of a letter token.

```
\token_if_other_p:N * \token_if_other_p:N <token>
\token_if_other:NTF * \token_if_other:NTF <token> {(true code)} {(false code)}
```

Tests if $\langle token \rangle$ has the category code of an “other” token.

```
\token_if_active_p:N * \token_if_active_p:N <token>
\token_if_active:NTF * \token_if_active:NTF <token> {(true code)} {(false code)}
```

Tests if $\langle token \rangle$ has the category code of an active character.

```
\token_if_eq_catcode_p:NN * \token_if_eq_catcode_p:NN <token1> <token2>
\token_if_eq_catcode:NNTF * \token_if_eq_catcode:NNTF <token1> <token2> {(true code)} {(false code)}
```

Tests if the two $\langle tokens \rangle$ have the same category code.

```
\token_if_eq_charcode_p:NN * \token_if_eq_charcode_p:NN <token1> <token2>
\token_if_eq_charcode:NNTF * \token_if_eq_charcode:NNTF <token1> <token2> {(true code)} {(false code)}
```

Tests if the two $\langle tokens \rangle$ have the same character code.

```
\token_if_eq_meaning_p:NN * \token_if_eq_meaning_p:NN <token1> <token2>
\token_if_eq_meaning:NNTF * \token_if_eq_meaning:NNTF <token1> <token2> {(true code)} {(false code)}
```

Tests if the two $\langle tokens \rangle$ have the same meaning when expanded.

```
\token_if_macro_p:N * \token_if_macro_p:N <token>
\token_if_macro:NTF * \token_if_macro:NTF <token> {(true code)} {(false code)}
```

Updated: 2011-05-23 Tests if the $\langle token \rangle$ is a \TeX macro.

```
\token_if_cs_p:N * \token_if_cs_p:N <token>
\token_if_cs:NTF * \token_if_cs:NTF <token> {(true code)} {(false code)}
```

Tests if the $\langle token \rangle$ is a control sequence.

```
\token_if_expandable_p:N * \token_if_expandable_p:N <token>
\token_if_expandable:NTF * \token_if_expandable:NTF <token> {(true code)} {(false code)}
```

Tests if the $\langle token \rangle$ is expandable. This test returns $\langle false \rangle$ for an undefined token.

```
\token_if_long_macro_p:N * \token_if_long_macro_p:N <token>
\token_if_long_macro:NTF * \token_if_long_macro:NTF <token> {(true code)} {(false code)}
```

Updated: 2012-01-20 Tests if the $\langle token \rangle$ is a long macro.

```
\token_if_protected_macro_p:N * \token_if_protected_macro_p:N <token>
\token_if_protected_macro:NTF * \token_if_protected_macro:NTF <token> {(true code)} {(false code)}
```

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**.

```

\token_if_protected_long_macro_p:N * \token_if_protected_long_macro_p:N <token>
\token_if_protected_long_macro:NTF * \token_if_protected_long_macro:NTF <token> {(true code)} {(false
code)}
Updated: 2012-01-20

```

Tests if the $\langle token \rangle$ is a protected long macro.

```

\token_if_chardef_p:N * \token_if_chardef_p:N <token>
\token_if_chardef:NTF * \token_if_chardef:NTF <token> {(true code)} {(false code)}
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 $\backslash chardefs$.

```

\token_if_mathchardef_p:N * \token_if_mathchardef_p:N <token>
\token_if_mathchardef:NTF * \token_if_mathchardef:NTF <token> {(true code)} {(false code)}
Updated: 2012-01-20

```

Tests if the $\langle token \rangle$ is defined to be a mathchardef.

```

\token_if_dim_register_p:N * \token_if_dim_register_p:N <token>
\token_if_dim_register:NTF * \token_if_dim_register:NTF <token> {(true code)} {(false code)}
Updated: 2012-01-20

```

Tests if the $\langle token \rangle$ is defined to be a dimension register.

```

\token_if_int_register_p:N * \token_if_int_register_p:N <token>
\token_if_int_register:NTF * \token_if_int_register:NTF <token> {(true code)} {(false code)}
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, $\backslash chardefs$, or $\backslash mathchardefs$ depending on their value.

```

\token_if_muskip_register_p:N * \token_if_muskip_register_p:N <token>
\token_if_muskip_register:NTF * \token_if_muskip_register:NTF <token> {(true code)} {(false code)}
New: 2012-02-15

```

Tests if the $\langle token \rangle$ is defined to be a muskip register.

```

\token_if_skip_register_p:N * \token_if_skip_register_p:N <token>
\token_if_skip_register:NTF * \token_if_skip_register:NTF <token> {(true code)} {(false code)}
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>*</code>	<code>\token_if_toks_register_p:N</code>	<code><token></code>
<code>\token_if_toks_register:NTF</code>	<code>*</code>	<code>\token_if_toks_register:NTF</code>	<code><token> {<true code>} {<false code>}</code>

Updated: 2012-01-20

Tests if the `<token>` is defined to be a toks register (not used by`LATEX3`).

<code>\token_if_primitive_p:N</code>	<code>*</code>	<code>\token_if_primitive_p:N</code>	<code><token></code>
<code>\token_if_primitive:NTF</code>	<code>*</code>	<code>\token_if_primitive:NTF</code>	<code><token> {<true code>} {<false code>}</code>

Updated: 2011-05-23

Tests if the `<token>` 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>	<code><function></code>	<code><token></code>
-----------------------------	-----------------------------	-------------------------------	----------------------------

Locally sets the test variable `\l_peek_token` equal to `<token>` (as an implicit token, *not* as a token list), and then expands the `<function>`. The `<token>` remains in the input stream as the next item after the `<function>`. The `<token>` here may be `␣`, `{` or `}` (assuming normal `TEX` 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>	<code><function></code>	<code><token></code>
------------------------------	------------------------------	-------------------------------	----------------------------

Globally sets the test variable `\g_peek_token` equal to `<token>` (as an implicit token, *not* as a token list), and then expands the `<function>`. The `<token>` remains in the input stream as the next item after the `<function>`. The `<token>` here may be `␣`, `{` or `}` (assuming normal `TEX` 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>	<code><test token></code>	<code>{<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 left in the input stream after the `<true code>` or `<false code>` (as appropriate to the result of the test).

`\peek_catcode_ignore_spaces:NTF` `\peek_catcode_ignore_spaces:NTF <test token> {(true code)} {(false 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).

`\peek_catcode_remove:NTF` `\peek_catcode_remove:NTF <test token> {(true code)} {(false 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).

`\peek_catcode_remove_ignore_spaces:NTF` `\peek_catcode_remove_ignore_spaces:NTF <test token> {(true code)} {(false 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).

`\peek_charcode:NTF` `\peek_charcode:NTF <test token> {(true code)} {(false 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).

`\peek_charcode_ignore_spaces:NTF` `\peek_charcode_ignore_spaces:NTF <test token> {(true code)} {(false 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).

`\peek_charcode_remove:NTF` `\peek_charcode_remove:NTF <test token> {(true code)} {(false 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:` (expanding to another internal token, `end of alignment template`) can be encountered when peeking ahead at the next token.
- 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

<properties> `macro`:*<parameters>*->*<replacement>*

where *<properties>* is among `\protected``\long``\outer`, *<parameters>* describes parameters that the macro expects, such as `#1#2#3`, and *<replacement>* 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`.

9 Internal functions

`_char_generate:nn` ★ `_char_generate:nn {<charcode>} {<catcode>}`

New: 2016-03-25

This function is identical in operation to the public `\char_generate:nn` but omits various sanity tests. In particular, this means it is used in certain places where engine variations need to be accounted for by the kernel. The *<catcode>* must give an explicit integer when expanded (and must not absorb a space for instance).

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:(NnV Nno Nnx NVn NVV Non Noo cnn cnV cno cnx cVn cVV con coo)</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>	<code>\prop_put:Nnn <property list></code> <code>{<key>} {<value>}</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:cnn</code> <code>\prop_gput_if_new:Nnn</code> <code>\prop_gput_if_new:cnn</code>	<code>\prop_put_if_new:Nnn <property list> {<key>} {<value>}</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:(NVN NoN cnN cVN coN)</code>	<code>\prop_get:NnN <property list> {<key>} <tl var></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:(NoN cnN coN)</code>	<code>\prop_pop:NnN <property list> {<key>} <tl var></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:(NoN cnN coN)</code>	<code>\prop_gpop:NnN <property list> {<key>} <tl var></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 <i>property list</i> \rangle $\{$ \langle <i>key</i> \rangle $\}$
<code>\prop_item:cn</code> *	Expands to the \langle <i>value</i> \rangle corresponding to the \langle <i>key</i> \rangle in the \langle <i>property list</i> \rangle . If the \langle <i>key</i> \rangle is missing, this has an empty expansion.

New: 2014-07-17

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:Nn</code> \langle <i>property list</i> \rangle $\{$ \langle <i>key</i> \rangle $\}$
<code>\prop_remove:(NV cn cV)</code>	Removes the entry listed under \langle <i>key</i> \rangle from the \langle <i>property list</i> \rangle . If the \langle <i>key</i> \rangle is not found in the \langle <i>property list</i> \rangle no change occurs, <i>i.e.</i> there is no need to test for the existence of a key before deleting it.
<code>\prop_gremove:Nn</code>	
<code>\prop_gremove:(NV cn cV)</code>	

New: 2012-05-12

5 Property list conditionals

<code>\prop_if_exist_p:N</code> *	<code>\prop_if_exist_p:N</code> \langle <i>property list</i> \rangle
<code>\prop_if_exist_p:c</code> *	<code>\prop_if_exist:NTF</code> \langle <i>property list</i> \rangle $\{$ \langle <i>true code</i> \rangle $\}$ $\{$ \langle <i>false code</i> \rangle $\}$
<code>\prop_if_exist:NTF</code> *	Tests whether the \langle <i>property list</i> \rangle is currently defined. This does not check that the \langle <i>property list</i> \rangle really is a property list variable.
<code>\prop_if_exist:cTF</code> *	

New: 2012-03-03

<code>\prop_if_empty_p:N</code> *	<code>\prop_if_empty_p:N</code> \langle <i>property list</i> \rangle
<code>\prop_if_empty_p:c</code> *	<code>\prop_if_empty:NTF</code> \langle <i>property list</i> \rangle $\{$ \langle <i>true code</i> \rangle $\}$ $\{$ \langle <i>false code</i> \rangle $\}$
<code>\prop_if_empty:NTF</code> *	Tests if the \langle <i>property list</i> \rangle is empty (containing no entries).
<code>\prop_if_empty:cTF</code> *	

<code>\prop_if_in_p:Nn</code>	*	<code>\prop_if_in:NnTF</code> \langle <i>property list</i> \rangle $\{$ \langle <i>key</i> \rangle $\}$ $\{$ \langle <i>true code</i> \rangle $\}$ $\{$ \langle <i>false code</i> \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.

```
\prop_get:NnNTF <property list> {<key>} <token list variable>
\prop_get:(NVN|NoN|cnN|cVN|coN)TF {<true code>} {<false code>}
```

Updated: 2012-05-19

If the *<key>* is not present in the *<property list>*, 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 *<key>* is present in the *<property list>*, stores the corresponding *<value>* in the *<token list variable>* without removing it from the *<property list>*, then leaves the *<true code>* in the input stream. The *<token list variable>* is assigned locally.

```
\prop_pop:NnNTF <property list> {<key>} <token list variable> {<true code>}
\prop_pop:cnNTF {<false code>}
```

New: 2011-08-18
Updated: 2012-05-19

If the *<key>* is not present in the *<property list>*, 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 *<key>* is present in the *<property list>*, pops the corresponding *<value>* in the *<token list variable>*, *i.e.* removes the item from the *<property list>*. Both the *<property list>* and the *<token list variable>* are assigned locally.

```
\prop_gpop:NnNTF <property list> {<key>} <token list variable> {<true code>}
\prop_gpop:cnNTF {<false code>}
```

New: 2011-08-18
Updated: 2012-05-19

If the *<key>* is not present in the *<property list>*, 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 *<key>* is present in the *<property list>*, pops the corresponding *<value>* in the *<token list variable>*, *i.e.* removes the item from the *<property list>*. The *<property list>* is modified globally, while the *<token list variable>* is assigned locally.

7 Mapping to property lists

```
\prop_map_function:NN ☆ <property list> <function>
```

```
\prop_map_function:cN ☆
```

Updated: 2013-01-08

Applies *<function>* to every *<entry>* stored in the *<property list>*. The *<function>* receives two argument for each iteration: the *<key>* and associated *<value>*. The order in which *<entries>* are returned is not defined and should not be relied upon.

```
\prop_map_inline:Nn <property list> {<inline function>}
```

```
\prop_map_inline:cn
```

Updated: 2013-01-08

Applies *<inline function>* to every *<entry>* stored within the *<property list>*. The *<inline function>* should consist of code which receives the *<key>* as #1 and the *<value>* as #2. The order in which *<entries>* are returned is not defined and should not be relied upon.

`\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 T_EX errors.

`\prop_map_break:n` ☆

Updated: 2012-06-29

`\prop_map_break:n {⟨tokens⟩}`

Used to terminate a `\prop_map_...` function before all entries in the *⟨property list⟩* have been processed, inserting the *⟨tokens⟩* 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 { <tokens> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\prop_map_...` scenario leads to low level T_EX errors.

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.

`\prop_log:N`

`\prop_log:c`

New: 2014-08-12

Updated: 2015-08-01

`\prop_log:N ⟨property list⟩`

Writes the entries in the *⟨property list⟩* in the log file.

9 Scratch property lists

`\l_tmpa_prop`
`\l_tmpb_prop`
New: 2012-06-23

Scratch property lists 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_prop`
`\g_tmpb_prop`
New: 2012-06-23

Scratch property lists 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.

10 Constants

`\c_empty_prop` A permanently-empty property list used for internal comparisons.

11 Internal property list functions

`\s__prop` The internal token used at the beginning of property lists. This is also used after each *key* (see `__prop_pair:wn`).

`__prop_pair:wn` `__prop_pair:wn <key> \s__prop {<item>}`

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.

`\l__prop_internal_tl` Token list used to store new key–value pairs to be inserted by functions of the `\prop_put:Nnn` family.

`__prop_split:NnTF`
Updated: 2013-01-08

`__prop_split:NnTF <property list> {<key>} {<true code>} {<false code>}`

Splits the *property list* at the *key*, giving three token lists: the *extract* of *property list* before the *key*, the *value* associated with the *key* and the *extract* of the *property list* after the *value*. Both *extracts* retain the internal structure of a property list, and the concatenation of the two *extracts* is a property list. If the *key* is present in the *property list* then the *true code* is left in the input stream, with #1, #2, and #3 replaced by the first *extract*, the *value*, and the second *extract*. If the *key* is not present in the *property list* then the *false code* is left in the input stream, with no trailing material. Both *true code* and *false code* are used in the replacement text of a macro defined internally, hence macro parameter characters should be doubled, except #1, #2, and #3 which stand in the *true code* for the three *extracts* from the property list. The *key* comparison takes place as described for `\str_if_eq:nn`.

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 *<message>* for the *<module>* 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 `on line`.

<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

`Fatal <module> error`

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.

<code>\msg_critical_text:n</code> *	<code>\msg_critical_text:n {<module>}</code>
-------------------------------------	--

Produces the standard text

`Critical <module> error`

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.

<code>\msg_error_text:n</code> *	<code>\msg_error_text:n {<module>}</code>
----------------------------------	---

Produces the standard text

`<module> error`

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.

<code>\msg_warning_text:n</code> *	<code>\msg_warning_text:n {<module>}</code>
------------------------------------	---

Produces the standard text

`<module> warning`

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_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 functions. After issuing a fatal error the `TEX` run halts.
`\msg_fatal:nnxxx`
`\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 functions. After issuing a critical error, `TEX` stops reading the current input file. This may halt the `TEX` run (if the current file is the main file) or may abort reading a sub-file.
`\msg_critical:nnxxx`
`\msg_critical:nnxx`
`\msg_critical:nnn`
`\msg_critical:nnx`
`\msg_critical:nn`

Updated: 2012-08-11

```

\msg_error:nnnnnn \msg_error:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
\msg_error:nnxxxx {\arg four}
\msg_error:nnnnn Issues {\module} error {\message}, passing {\arg one} to {\arg four} to the text-creating
\msg_error:nnxxx functions. The error interrupts processing and issues the text at the terminal. After user
\msg_error:nnnn input, the run continues.
\msg_error:nnxx
\msg_error:nnn
\msg_error:nnx
\msg_error:nn

```

Updated: 2012-08-11

```

\msg_warning:nnnnnn \msg_warning:nnxxxx {\module} {\message} {\arg one} {\arg two} {\arg three}
\msg_warning:nnxxxx {\arg four}
\msg_warning:nnnnn Issues {\module} warning {\message}, passing {\arg one} to {\arg four} to the text-creating
\msg_warning:nnxxx functions. The warning text is added to the log file and the terminal, but the TEX run
\msg_warning:nnnn is not interrupted.
\msg_warning:nnxx
\msg_warning:nnn
\msg_warning:nnx
\msg_warning:nn

```

Updated: 2012-08-11

```

\msg_info:nnnnnn \msg_info:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_info:nnxxxx four}
\msg_info:nnnnn Issues {\module} information {\message}, passing {\arg one} to {\arg four} to the text-creating
\msg_info:nnxxx functions. The information text is added to the log file.
\msg_info:nnnn
\msg_info:nnxx
\msg_info:nnn
\msg_info:nnx
\msg_info:nn

```

Updated: 2012-08-11

```

\msg_log:nnnnnn \msg_log:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_log:nnxxxx four}
\msg_log:nnnnn Issues {\module} information {\message}, passing {\arg one} to {\arg four} to the text-creating
\msg_log:nnxxx functions. The information text is added to the log file: the output is briefer than \msg_
\msg_log:nnnn info:nnnnnn.
\msg_log:nnxx
\msg_log:nnn
\msg_log:nnx
\msg_log:nn

```

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_redirect_module:nnn`

Updated: 2012-04-27

`\msg_redirect_module:nnn {<module>} {<class one>} {<class two>}`

Redirects message of *<class one>* for *<module>* to act as though they were from *<class two>*. Messages of *<class one>* from sources other than *<module>* are not affected by this redirection. This function can be used to make some messages “silent” by default. For example, all of the `warning` messages of *<module>* could be turned off with:

```
\msg_redirect_module:nnn { module } { warning } { none }
```

`\msg_redirect_name:nnn`

Updated: 2012-04-27

`\msg_redirect_name:nnn {<module>} {<message>} {<class>}`

Redirects a specific *<message>* from a specific *<module>* to act as a member of *<class>* of messages. No further redirection is performed. This function can be used to make a selected message “silent” without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

5 Low-level message functions

The lower-level message functions should usually be accessed from the higher-level system. However, there are occasions where direct access to these functions is desirable.

`\msg_interrupt:nnn`

New: 2012-06-28

`\msg_interrupt:nnn {<first line>} {<text>} {<extra text>}`

Interrupts the \TeX run, issuing a formatted message comprising *<first line>* and *<text>* laid out in the format

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! <first line>
!
! <text>
!.....
```

where the *<text>* is wrapped to fit within the current line length. The user may then request more information, at which stage the *<extra text>* is shown in the terminal in the format

```
|,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
| <extra text>
|.....
```

where the *<extra text>* is wrapped within the current line length. Wrapping of both *<text>* and *<more text>* takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

`\msg_log:n` `\msg_log:n {<text>}`

New: 2012-06-28 Writes to the log file with the `<text>` laid out in the format

```
.....  
. <text>  
.....
```

where the `<text>` 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 `<text>` laid out in the format

```
*****  
* <text>  
*****
```

where the `<text>` 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.

6 Kernel-specific functions

Messages from L^AT_EX₃ itself are handled by the general message system, but have their own functions. This allows some text to be pre-defined, and also ensures that serious errors can be handled properly.

`_msg_kernel_new:nnnn` `_msg_kernel_new:nnnn {<module>} {<message>} {<text>} {<more text>}`

`_msg_kernel_new:nnn`

Updated: 2011-08-16

Creates a kernel `<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 and expanded at the time the message is used. An error is raised if the `<message>` already exists.

`_msg_kernel_set:nnnn` `_msg_kernel_set:nnnn {<module>} {<message>} {<text>} {<more text>}`

`_msg_kernel_set:nnn`

Sets up the text for a kernel `<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 and expanded at the time the message is used.

```
\_msg_kernel_fatal:nnnnnn
\_msg_kernel_fatal:nnxxxx
\_msg_kernel_fatal:nnnnn
\_msg_kernel_fatal:nnxxx
\_msg_kernel_fatal:nnnn
\_msg_kernel_fatal:nnxx
\_msg_kernel_fatal:nnn
\_msg_kernel_fatal:nnx
\_msg_kernel_fatal:nn
```

Updated: 2012-08-11

```
\_msg_kernel_fatal:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg
three>} {<arg four>}
```

Issues kernel *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a fatal error the T_EX run halts. Cannot be redirected.

```
\_msg_kernel_error:nnnnnn
\_msg_kernel_error:nnxxxx
\_msg_kernel_error:nnnnn
\_msg_kernel_error:nnxxx
\_msg_kernel_error:nnnn
\_msg_kernel_error:nnxx
\_msg_kernel_error:nnn
\_msg_kernel_error:nnx
\_msg_kernel_error:nn
```

Updated: 2012-08-11

```
\_msg_kernel_error:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg
three>} {<arg four>}
```

Issues kernel *<module>* error *<message>*, passing *<arg one>* to *<arg four>* 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.

```
\_msg_kernel_warning:nnnnnn
\_msg_kernel_warning:nnxxxx
\_msg_kernel_warning:nnnnn
\_msg_kernel_warning:nnxxx
\_msg_kernel_warning:nnnn
\_msg_kernel_warning:nnxx
\_msg_kernel_warning:nnn
\_msg_kernel_warning:nnx
\_msg_kernel_warning:nn
```

Updated: 2012-08-11

```
\_msg_kernel_warning:nnnnnn {<module>} {<message>} {<arg one>} {<arg
two>} {<arg three>} {<arg four>}
```

Issues kernel *<module>* warning *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The warning text is added to the log file, but the T_EX run is not interrupted.

```
\_msg_kernel_info:nnnnnn
\_msg_kernel_info:nnxxxx
\_msg_kernel_info:nnnnn
\_msg_kernel_info:nnxxx
\_msg_kernel_info:nnnn
\_msg_kernel_info:nnxx
\_msg_kernel_info:nnn
\_msg_kernel_info:nnx
\_msg_kernel_info:nn
```

Updated: 2012-08-11

```
\_msg_kernel_info:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>} {<arg
three>} {<arg four>}
```

Issues kernel *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The information text is added to the log file.

7 Expandable errors

In a few places, the L^AT_EX3 kernel needs to produce errors in an expansion only context. This 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. However, the interface is similar, with the important caveat that the message text and arguments are not expanded, and messages should be very short.

```
\_msg_kernel_expandable_error:nnnnnn * \_msg_kernel_expandable_error:nnnnnn {<module>} {<message>}
\_msg_kernel_expandable_error:nnnnnn * {<arg one>} {<arg two>} {<arg three>} {<arg four>}
\_msg_kernel_expandable_error:nnnnn *
\_msg_kernel_expandable_error:nnnn *
\_msg_kernel_expandable_error:nnn *
\_msg_kernel_expandable_error:nn *
\_msg_kernel_expandable_error:nn *
```

New: 2011-11-23

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.

```
\_msg_expandable_error:n * \_msg_expandable_error:n {<error message>}
```

New: 2011-08-11

Updated: 2011-08-13

Issues an “Undefined error” message from T_EX itself, and prints the *<error message>*. The *<error message>* must be short: it is cropped at the end of one line.

T_EXhackers note: This function expands to an empty token list after two steps. Tokens inserted in response to T_EX’s prompt are read with the current category code setting, and inserted just after the place where the error message was issued.

8 Internal l3msg functions

The following functions are used in several kernel modules.

```
\_msg_log_next: \_msg_log_next: <show-command>
```

New: 2015-08-05

Causes the next *<show-command>* to send its output to the log file instead of the terminal. This allows for instance `\cs_log:N` to be defined as `_msg_log_next: \cs_show:N`. The effect of this command lasts until the next use of `_msg_show_wrap:Nn` or `_msg_show_wrap:n` or `_msg_show_variable:NNNnn`, in other words until the next time the ε -T_EX primitive `\showtokens` would have been used for showing to the terminal or until the next `variable-not-defined` error.

```
\_msg_show_pre:nnnnnn \_msg_show_pre:nnnnnn {<module>} {<message>} {<arg one>} {<arg two>}
\_msg_show_pre:(nnxxxx|nnnnnV) {<arg three>} {<arg four>}
```

New: 2015-08-05

Prints the *<message>* from *<module>* in the terminal (or log file if `_msg_log_next:` was issued) without formatting. Used in messages which print complex variable contents completely.

`_msg_show_variable:NNNnn`
New: 2015-08-04

`_msg_show_variable:NNNnn` $\langle variable \rangle$ $\langle if-exist \rangle$ $\langle if-empty \rangle$ $\{\langle msg \rangle\}$ $\{\langle formatted content \rangle\}$

If the $\langle variable \rangle$ does not exist according to $\langle if-exist \rangle$ (typically `\cs_if_exist:NTF`) then throw an error and do nothing more. Otherwise, if $\langle msg \rangle$ is not empty, display the message `LaTeX/kernel/show- $\langle msg \rangle$` with `\token_to_str:N` $\langle variable \rangle$ as a first argument, and a second argument that is `?` or empty depending on the result of $\langle if-empty \rangle$ (typically `\tl_if_empty:NTF`) on the $\langle variable \rangle$. Then display the $\langle formatted content \rangle$ by giving it as an argument to `_msg_show_wrap:n`.

`_msg_show_wrap:Nn`
New: 2015-08-03
Updated: 2015-08-07

`_msg_show_wrap:Nn` $\langle function \rangle$ $\{\langle expression \rangle\}$

Shows or logs the $\langle expression \rangle$ (turned into a string), an equal sign, and the result of applying the $\langle function \rangle$ to the $\{\langle expression \rangle\}$. For instance, if the $\langle function \rangle$ is `\int_eval:n` and the $\langle expression \rangle$ is `1+2` then this logs `> 1+2=3`. The case where the $\langle function \rangle$ is `\tl_to_str:n` is special: then the string representation of the $\langle expression \rangle$ is only logged once.

`_msg_show_wrap:n`
New: 2015-08-03

`_msg_show_wrap:n` $\{\langle formatted text \rangle\}$

Shows or logs the $\langle formatted text \rangle$. After expansion, unless it is empty, the $\langle formatted text \rangle$ must contain `>`, and the part of $\langle formatted text \rangle$ before the first `>` is removed. Failure to do so causes low-level TeX errors.

`_msg_show_item:n`
`_msg_show_item:nn`
`_msg_show_item_unbraced:nn`
Updated: 2012-09-09

`_msg_show_item:n` $\{\langle item \rangle\}$
`_msg_show_item:nn` $\{\langle item-key \rangle\}$ $\{\langle item-value \rangle\}$

Auxiliary functions used within the last argument of `_msg_show_variable:NNNnn` or `_msg_show_wrap:n` to format variable items correctly for display. The `_msg_show_item:n` version is used for simple lists, the `_msg_show_item:nn` and `_msg_show_item_unbraced:nn` versions for key-value like data structures.

`\c_msg_coding_error_text_tl`

The text

This is a coding error.

used by kernel functions when erroneous programming input is encountered.

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 \TeX attempts to locate them using both the operating system path and entries in the \TeX file database (most \TeX systems use such a database). Thus the “current path” for \TeX is somewhat broader than that for other programs.

For functions which expect a $\langle file\ name \rangle$ 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 File operation functions

`\g_file_curr_dir_str`
`\g_file_curr_name_str`
`\g_file_curr_ext_str`

New: 2017-06-21

Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (*i.e.* if it is in the \TeX search path), and does *not* end in `/` other than the case that it is exactly equal to the root directory. The $\langle name \rangle$ and $\langle ext \rangle$ parts together make up the file name, thus the $\langle name \rangle$ part may be thought of as the “job name” for the current file. Note that \TeX does not provide information on the $\langle ext \rangle$ part for the main (top level) file and that this file always has an empty $\langle dir \rangle$ component. Also, the $\langle name \rangle$ here will be equal to `\c_sys_jobname_str`, which may be different from the real file name (if set using `--jobname`, for example).

`\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.

\TeX hackers note: When working as a package in $\LaTeX_{2\epsilon}$, `expl3` will automatically append the current `\input@path` to the set of values from `\l_file_search_path_seq`.

`\file_if_exist:nTF` $\langle file\ name \rangle$ $\langle true\ code \rangle$ $\langle false\ code \rangle$

Updated: 2012-02-10

Searches for $\langle file\ name \rangle$ using the current \TeX 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` $\{\langle file\ name\rangle\}$ $\langle str\ var\rangle$

Searches for $\langle file\ name\rangle$ in the path as detailed for `\file_if_exist:nTF`, and if found sets the $\langle str\ var\rangle$ the fully-qualified name of the file, *i.e.* the path and file name. This includes an extension `.tex` when the given $\langle file\ name\rangle$ has no extension but the file found has that extension. If the file is not found then the $\langle str\ var\rangle$ is empty.

`\file_parse_full_name:nNNN`

New: 2017-06-23

Updated: 2017-06-26

`\file_parse_full_name:nNNN` $\{\langle full\ name\rangle\}$ $\langle dir\rangle$ $\langle name\rangle$ $\langle ext\rangle$

Parses the $\langle full\ name\rangle$ and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The $\langle dir\rangle$: everything up to the last `/` (path separator) in the $\langle file\ path\rangle$. As with system `PATH` variables and related functions, the $\langle dir\rangle$ does *not* include the trailing `/` unless it points to the root directory. If there is no path (only a file name), $\langle dir\rangle$ is empty.
- The $\langle name\rangle$: everything after the last `/` up to the last `.`, where both of those characters are optional. The $\langle name\rangle$ may contain multiple `.` characters. It is empty if $\langle full\ name\rangle$ consists only of a directory name.
- The $\langle ext\rangle$: everything after the last `.` (including the dot). The $\langle ext\rangle$ is empty if there is no `.` after the last `/`.

This function does not expand the $\langle full\ name\rangle$ before turning it to a string. It assume that the $\langle full\ name\rangle$ either contains no quote (`"`) characters or is surrounded by a pair of quotes.

`\file_input:n`

Updated: 2017-06-26

`\file_input:n` $\{\langle file\ name\rangle\}$

Searches for $\langle file\ name\rangle$ 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.

1.1 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_EX 3. 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:c</code> <code>\iow_new:N</code> <code>\iow_new:c</code>	<code>\ior_new:N <stream></code> <code>\iow_new:N <stream></code> Globally reserves the name of the <code><stream></code> , either for reading or for writing as appropriate. The <code><stream></code> is not opened until the appropriate <code>\..._open:Nn</code> function is used. Attempting to use a <code><stream></code> which has not been opened is an error, and the <code><stream></code> will behave as the corresponding <code>\c_term_...</code>
New: 2011-09-26 Updated: 2011-12-27	
<code>\ior_open:Nn</code> <code>\ior_open:cn</code>	<code>\ior_open:Nn <stream> {<file name>}</code> Opens <code><file name></code> for reading using <code><stream></code> as the control sequence for file access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><file name></code> 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.
Updated: 2012-02-10	
<code>\ior_open:NnTF</code> <code>\ior_open:cnTF</code>	<code>\ior_open:NnTF <stream> {<file name>} {<>true code>} {<>false code>}</code> Opens <code><file name></code> for reading using <code><stream></code> as the control sequence for file access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><file name></code> until a <code>\ior_close:N</code> instruction is given or the T _E X run ends. The <code><>true code></code> is then inserted into the input stream. If the file is not found, no error is raised and the <code><>false code></code> is inserted into the input stream.
New: 2013-01-12	
<code>\iow_open:Nn</code> <code>\iow_open:cn</code>	<code>\iow_open:Nn <stream> {<file name>}</code> Opens <code><file name></code> for writing using <code><stream></code> as the control sequence for file access. If the <code><stream></code> was already open it is closed before the new operation begins. The <code><stream></code> is available for access immediately and will remain allocated to <code><file name></code> 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).
Updated: 2012-02-09	
<code>\ior_close:N</code> <code>\ior_close:c</code> <code>\iow_close:N</code> <code>\iow_close:c</code>	<code>\ior_close:N <stream></code> <code>\iow_close:N <stream></code> Closes the <code><stream></code> . Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.
Updated: 2012-07-31	
<code>\ior_show_list:</code> <code>\ior_log_list:</code> <code>\iow_show_list:</code> <code>\iow_log_list:</code>	<code>\ior_show_list:</code> <code>\ior_log_list:</code> <code>\iow_show_list:</code> <code>\iow_log_list:</code> 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.
New: 2017-06-27	

1.2 Reading from files

`\ior_get:NN` `\ior_get:NN` $\langle stream \rangle$ $\langle token list variable \rangle$

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 $\langle stream \rangle$ and stores the result locally in the $\langle token list \rangle$ variable. If the $\langle stream \rangle$ is not open, input is requested from the terminal. The material read from the $\langle stream \rangle$ is tokenized by \TeX 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
\l_set:Nn \l_tmpb_tl { \par }
\l_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.

\TeX hackers note: This protected macro is a wrapper around the \TeX primitive `\read`. Regardless of settings, \TeX 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.

`\ior_str_get:NN` `\ior_str_get:NN` $\langle stream \rangle$ $\langle token list variable \rangle$

New: 2016-12-04

Function that reads one line from the input $\langle stream \rangle$ and stores the result locally in the $\langle token list \rangle$ variable. If the $\langle stream \rangle$ is not open, input is requested from the terminal. The material is read from the $\langle stream \rangle$ 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 $\langle token list variable \rangle$ 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.

\TeX hackers note: This protected macro is a wrapper around the ε - \TeX primitive `\readline`. Regardless of settings, \TeX 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`.

`\ior_map_inline:Nn`

New: 2012-02-11

`\ior_map_inline:Nn <stream> {<inline function>}`

Applies the *<inline function>* to each set of *<lines>* obtained by calling `\ior_get:NN` until reaching the end of the file. T_EX ignores any trailing new-line marker from the file it reads. The *<inline function>* should consist of code which receives the *<line>* as #1.

`\ior_str_map_inline:Nn`

New: 2012-02-11

`\ior_str_map_inline:Nn <stream> {<inline function>}`

Applies the *<inline function>* to every *<line>* in the *<stream>*. 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). The *<inline function>* should consist of code which receives the *<line>* as #1. Note that T_EX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T_EX also ignores any trailing new-line marker from the file it reads.

`\ior_map_break:`

New: 2012-06-29

`\ior_map_break:`

Used to terminate a `\ior_map...` function before all lines from the *<stream>* have been processed. 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: }
  {
    % Do something useful
  }
}
```

Use outside of a `\ior_map...` scenario leads to low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before further items are taken from the input stream. This depends on the design of the mapping function.

`\ior_map_break:n`

New: 2012-06-29

`\ior_map_break:n` $\{(tokens)\}$

Used to terminate a `\ior_map_...` function before all lines in the $\langle stream \rangle$ have been processed, inserting the $\langle tokens \rangle$ 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 { <tokens> } }
  {
    % Do something useful
  }
}
```

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

\TeX hackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `__prg_break_point:Nn` before the $\langle tokens \rangle$ are inserted into the input stream. This depends on the design of the mapping function.

`\ior_if_eof_p:N` \star

`\ior_if_eof:NTF` \star

Updated: 2012-02-10

`\ior_if_eof_p:N` $\langle stream \rangle$

`\ior_if_eof:NTF` $\langle stream \rangle$ $\{(true\ code)\}$ $\{(false\ code)\}$

Tests if the end of a $\langle stream \rangle$ has been reached during a reading operation. The test also returns a true value if the $\langle stream \rangle$ is not open.

2 Writing to files

`\iow_now:Nn`

`\iow_now:(Nx|cn|cx)`

Updated: 2012-06-05

`\iow_now:Nn` $\langle stream \rangle$ $\{(tokens)\}$

This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ 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 $\langle tokens \rangle$ 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 $\langle tokens \rangle$ to the terminal file immediately: it is a dedicated version of `\iow_now:Nn`.

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

`\iow_shipout:Nn <stream> {<tokens>}`

This functions writes `<tokens>` to the specified `<stream>` when the current page is finalised (*i.e.* at shipout). The `x`-type variants expand the `<tokens>` at the point where the function is used but *not* when the resulting tokens are written to the `<stream>` (*cf.* `\iow_shipout_x:Nn`).

TeXhackers note: When using `expl3` with a format other than \LaTeX , new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

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

`\iow_shipout_x:Nn <stream> {<tokens>}`

This functions writes `<tokens>` to the specified `<stream>` when the current page is finalised (*i.e.* at shipout). The `<tokens>` 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).

Updated: 2012-09-08

TeXhackers note: This is a wrapper around the \TeX primitive `\write`. When using `expl3` with a format other than \LaTeX , new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

`\iow_char:N` ★

`\iow_char:N \<char>`

Inserts `<char>` into the output stream. Useful when trying to write difficult characters such as `%`, `{`, `}`, *etc.* in messages, for example:

```
\iow_now:Nx \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

`\iow_newline:` ★

`\iow_newline:`

Function to add a new line within the `<tokens>` written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

TeXhackers note: When using `expl3` with a format other than \LaTeX , the character inserted by `\iow_newline:` is not recognized by \TeX , which may lead to the insertion of additional unwanted line-breaks. This issue only affects `\iow_shipout:Nn`, `\iow_shipout_x:Nn` and direct uses of primitive operations.

2.1 Wrapping lines in output

`\iow_wrap:nnnN`

New: 2012-06-28

Updated: 2017-07-17

`\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:

- `\` 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.

TeXhackers 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 TeX system in use: the standard value is 78, which is typically correct for unmodified TeXlive and MiKTeX systems.

2.2 Constant input–output streams

`\c_term_ior` Constant input stream for reading from the terminal. Reading from this stream using `\ior_get:NN` or similar results in a prompt from T_EX of the form

`<tl>=`

`\c_log_ior` Constant output streams for writing to the log and to the terminal (plus the log), respectively.
`\c_term_ior`

2.3 Primitive conditionals

`\if_eof:w` ★ `\if_eof:w <stream>`
`<true code>`
`\else:`
`<false code>`
`\fi:`

Tests if the `<stream>` returns “end of file”, which is true for non-existent files. The `\else:` branch is optional.

T_EXhackers note: This is the T_EX primitive `\ifeof`.

2.4 Internal file functions and variables

`\g__file_internal_ior` Used to test for the existence of files when opening.

`\l__file_base_name_str` Used to store and transfer the file name (including extension) and (partial) file path
`\l__file_full_name_str` whilst reading files. (The file base is the base name plus any preceding directory name.)

`__file_missing:n` `__file_missing:n {<name>}`

New: 2017-06-25

Expands the `<name>` as per `__file_name_sanitize:nN` then produces an error message indicating that that file was not found.

`__file_name_sanitize:nN` `__file_name_sanitize:nN {<name>} <str var>`

New: 2017-06-19

Exhaustively-expands the `<name>` with the exception of any category `<active>` (catcode 13) tokens, which are not expanded. The list of `<active>` tokens is taken from `\l_char_active_seq`. The `<str var>` is then set to the `<sanitized name>`.

`__file_name_quote:nN` `__file_name_quote:nN {<name>} <str var>`

New: 2017-06-19
Updated: 2017-06-25

Expands the `<name>` (without special-casing active tokens), then sets the `<str var>` to the `<name>` quoted using " at each end if required by the presence of spaces in the `<name>`. Any existing " tokens is removed and if their number is odd an error is raised.

2.5 Internal input–output functions

`_ior_open:Nn` `_ior_open:Nn` $\langle stream \rangle$ $\{\langle file\ name \rangle\}$

`_ior_open:No`

New: 2012-01-23

This function has identical syntax to the public version. However, it does not take precautions against active characters in the $\langle file\ name \rangle$, and it does not attempt to add a $\langle path \rangle$ to the $\langle file\ name \rangle$: it is therefore intended to be used by higher-level functions which have already fully expanded the $\langle file\ name \rangle$ and which need to perform multiple open or close operations. See for example the implementation of `\file_get_full_name:nN`,

`_iow_with:Nnn` `_iow_with:Nnn` $\langle integer \rangle$ $\{\langle value \rangle\}$ $\{\langle code \rangle\}$

New: 2014-08-23

If the $\langle integer \rangle$ is equal to the $\langle value \rangle$ then this function simply runs the $\langle code \rangle$. Otherwise it saves the current value of the $\langle integer \rangle$, sets it to the $\langle value \rangle$, runs the $\langle code \rangle$, and restores the $\langle integer \rangle$ to its former value. This is used to ensure that the `\newlinechar` is 10 when writing to a stream, which lets `\iow_newline:` work, and that `\errorcontextlines` is -1 when displaying a message.

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

`\dim_new:N` `\dim_new:N` $\langle dimension \rangle$

`\dim_new:c`

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.

`\dim_const:Nn` `\dim_const:Nn` $\langle dimension \rangle$ $\{ \langle dimension expression \rangle \}$

`\dim_const:cn`

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

`\dim_zero:N` `\dim_zero:N` $\langle dimension \rangle$

`\dim_zero:c`

Sets $\langle dimension \rangle$ to 0pt.

`\dim_gzero:N`

`\dim_gzero:c`

`\dim_zero_new:N` `\dim_zero_new:N` $\langle dimension \rangle$

`\dim_zero_new:c`

`\dim_gzero_new:N`

`\dim_gzero_new:c`

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

`\dim_if_exist_p:N` ★ `\dim_if_exist_p:N` $\langle dimension \rangle$

`\dim_if_exist_p:c` ★ `\dim_if_exist:NTF` $\langle dimension \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$

`\dim_if_exist:NTF` ★ Tests whether the $\langle dimension \rangle$ is currently defined. This does not check that the

`\dim_if_exist:cTF` ★ $\langle dimension \rangle$ 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 <i><dimension expression></i> to the current content of the <i><dimension></i> .
<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 <i><dimension></i> to the value of <i><dimension expression></i> , 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 <dimension₁₂></code>
<code>\dim_set_eq:(cN Nc cc)</code>	Sets the content of <i><dimension₁></i> equal to that of <i><dimension₂></i> .
<code>\dim_gset_eq:NN</code>	
<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 <i><dimension expression></i> from the current content of the <i><dimension></i> .
<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 <i><dimexpr></i> to its absolute value, leaving the result in the input stream as a <i><dimension denotation></i> .

<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 <i><dimension expressions></i> and leaves either the maximum or minimum value in the input stream as appropriate, as a <i><dimension denotation></i> .

`\dim_ratio:nn` ☆

`\dim_ratio:nn {⟨dimexpr1⟩} {⟨dimexpr2⟩}`

Updated: 2011-10-22

Parses the two *⟨dimension expressions⟩* and converts the ratio of the two to a form suitable for use inside a *⟨dimension expression⟩*. 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_p:nNn {⟨dimexpr1⟩} ⟨relation⟩ {⟨dimexpr2⟩}`

`\dim_compare:nNnTF` ☆

`\dim_compare:nNnTF`
`{⟨dimexpr1⟩} ⟨relation⟩ {⟨dimexpr2⟩}`
`{⟨true code⟩} {⟨false code⟩}`

This function first evaluates each of the *⟨dimension expressions⟩* as described for `\dim_eval:n`. The two results are then compared using the *⟨relation⟩*:

Equal	=
Greater than	>
Less than	<

```

\dim_compare_p:n * \dim_compare_p:n
\dim_compare:nTF * {
    <dimexpr1> <relation1>
    ...
    <dimexprN> <relationN>
    <dimexprN+1>
}
\dim_compare:nTF
{
    <dimexpr1> <relation1>
    ...
    <dimexprN> <relationN>
    <dimexprN+1>
}
{(true code)} {(false code)}

```

Updated: 2013-01-13

This function evaluates the *<dimension expressions>* as described for `\dim_eval:n` and compares consecutive result using the corresponding *<relation>*, namely it compares *<dimexpr₁>* and *<dimexpr₂>* using the *<relation₁>*, then *<dimexpr₂>* and *<dimexpr₃>* using the *<relation₂>*, until finally comparing *<dimexpr_N>* and *<dimexpr_{N+1}>* using the *<relation_N>*. The test yields `true` if all comparisons are `true`. Each *<dimension expression>* is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other *<dimension 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	!=

```

\dim_case:nn ☆ \dim_case:nnTF {⟨test dimension expression⟩}
\dim_case:nnTF ☆ {
  {⟨dimexpr case1⟩} {⟨code case1⟩}
  {⟨dimexpr case2⟩} {⟨code case2⟩}
  ...
  {⟨dimexpr casen⟩} {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false 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

```

\dim_do_until:nNnn ☆ \dim_do_until:nNnn {⟨dimexpr1⟩} ⟨relation⟩ {⟨dimexpr2⟩} {⟨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**.

```

\dim_do_while:nNnn ☆ \dim_do_while:nNnn {⟨dimexpr1⟩} ⟨relation⟩ {⟨dimexpr2⟩} {⟨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**.

```

\dim_until_do:nNnn ☆ \dim_until_do:nNnn {⟨dimexpr1⟩} ⟨relation⟩ {⟨dimexpr2⟩} {⟨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**.

`\dim_while_do:nNnn` ☆ `\dim_while_do:nNnn {⟨dimexpr1⟩} ⟨relation⟩ {⟨dimexpr2⟩} {⟨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 **true**. After the *⟨code⟩* has been processed by \TeX the test is repeated, and a loop occurs until the test is **false**.

`\dim_do_until:nn` ☆ `\dim_do_until:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Places the *⟨code⟩* in the input stream for \TeX to process, and then evaluates the *⟨dimension relation⟩* as described for `\dim_compare:nTF`. If the test is **false** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **true**.

`\dim_do_while:nn` ☆ `\dim_do_while:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Places the *⟨code⟩* in the input stream for \TeX to process, and then evaluates the *⟨dimension relation⟩* as described for `\dim_compare:nTF`. If the test is **true** then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is **false**.

`\dim_until_do:nn` ☆ `\dim_until_do:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Evaluates the *⟨dimension relation⟩* as described for `\dim_compare:nTF`, and then places the *⟨code⟩* in the input stream if the *⟨relation⟩* is **false**. After the *⟨code⟩* has been processed by \TeX the test is repeated, and a loop occurs until the test is **true**.

`\dim_while_do:nn` ☆ `\dim_while_do:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Evaluates the *⟨dimension relation⟩* as described for `\dim_compare:nTF`, and then places the *⟨code⟩* in the input stream if the *⟨relation⟩* is **true**. After the *⟨code⟩* has been processed by \TeX the test is repeated, and a loop occurs until the test is **false**.

6 Using dim expressions and variables

`\dim_eval:n` ☆ `\dim_eval:n {⟨dimension expression⟩}`

Updated: 2011-10-22

Evaluates the *⟨dimension expression⟩*, expanding any dimensions and token list variables within the *⟨expression⟩* 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 *⟨dimension denotation⟩* 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 *⟨internal dimension⟩*.

`\dim_use:N` ☆ `\dim_use:N ⟨dimension⟩`

`\dim_use:c` ☆

Recovers the content of a *⟨dimension⟩* 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 *⟨dimension⟩* 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` ★ `\dim_to_decimal:n {⟨dimexpr⟩}`

New: 2014-07-15

Evaluates the *⟨dimension expression⟩*, 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.

`\dim_to_decimal_in_bp:n` ★ `\dim_to_decimal_in_bp:n {⟨dimexpr⟩}`

New: 2014-07-15

Evaluates the *⟨dimension expression⟩*, and leaves the result, expressed in big points (`bp`) 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_in_bp:n { 1pt }
```

leaves 0.99628 in the input stream, *i.e.* the magnitude of one (\TeX) point when converted to big points.

`\dim_to_decimal_in_sp:n` ★ `\dim_to_decimal_in_sp:n {⟨dimexpr⟩}`

New: 2015-05-18

Evaluates the *⟨dimension expression⟩*, and leaves the result, expressed in scaled points (`sp`) in the input stream, with *no units*. The result is necessarily an integer.

`\dim_to_decimal_in_unit:nn` ★ `\dim_to_decimal_in_unit:nn {⟨dimexpr1⟩} {⟨dimexpr2⟩}`

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 \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_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 - \TeX primitives, which is required internally by `\dim_to_decimal_in_unit:nn`.

<code>\dim_to_fp:n</code> <small>★</small>	<code>\dim_to_fp:n {⟨<i>dimexpr</i>⟩}</code>
<small>New: 2012-05-08</small>	Expands to an internal floating point number equal to the value of the $\langle \textit{dimexpr} \rangle$ 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.

7 Viewing dim variables

<code>\dim_show:N</code> <code>\dim_show:c</code>	<code>\dim_show:N ⟨<i>dimension</i>⟩</code>
	Displays the value of the $\langle \textit{dimension} \rangle$ on the terminal.

<code>\dim_show:n</code>	<code>\dim_show:n {⟨<i>dimension expression</i>⟩}</code>
<small>New: 2011-11-22 Updated: 2015-08-07</small>	Displays the result of evaluating the $\langle \textit{dimension expression} \rangle$ on the terminal.

<code>\dim_log:N</code> <code>\dim_log:c</code>	<code>\dim_log:N ⟨<i>dimension</i>⟩</code>
<small>New: 2014-08-22 Updated: 2015-08-03</small>	Writes the value of the $\langle \textit{dimension} \rangle$ in the log file.

<code>\dim_log:n</code>	<code>\dim_log:n {⟨<i>dimension expression</i>⟩}</code>
<small>New: 2014-08-22 Updated: 2015-08-07</small>	Writes the result of evaluating the $\langle \textit{dimension expression} \rangle$ in the log file.

8 Constant dimensions

<code>\c_max_dim</code>	The maximum value that can be stored as a dimension. This can also be used as a component of a skip.
-------------------------	--

<code>\c_zero_dim</code>	A zero length as a dimension. This can also be used as a component of a skip.
--------------------------	---

9 Scratch dimensions

<code>\l_tmpa_dim</code> <code>\l_tmpb_dim</code>	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.
--	---

<code>\g_tmpa_dim</code> <code>\g_tmpb_dim</code>	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.
--	--

10 Creating and initialising skip variables

`\skip_new:N`
`\skip_new:c`

`\skip_new:N` $\langle skip \rangle$

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 0pt.

`\skip_const:Nn`
`\skip_const:cn`

`\skip_const:Nn` $\langle skip \rangle$ $\{ \langle skip \text{ expression} \rangle \}$

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 $\langle skip \text{ expression} \rangle$.

New: 2012-03-05

`\skip_zero:N`
`\skip_zero:c`
`\skip_gzero:N`
`\skip_gzero:c`

`\skip_zero:N` $\langle skip \rangle$

Sets $\langle skip \rangle$ to 0pt.

`\skip_zero_new:N`
`\skip_zero_new:c`
`\skip_gzero_new:N`
`\skip_gzero_new:c`

`\skip_zero_new:N` $\langle skip \rangle$

Ensures that the $\langle skip \rangle$ exists globally by applying `\skip_new:N` if necessary, then applies `\skip_(g)zero:N` to leave the $\langle skip \rangle$ set to zero.

New: 2012-01-07

`\skip_if_exist_p:N` *
`\skip_if_exist_p:c` *
`\skip_if_exist:NTF` *
`\skip_if_exist:cTF` *

`\skip_if_exist_p:N` $\langle skip \rangle$

`\skip_if_exist:NTF` $\langle skip \rangle$ $\{ \langle true \text{ code} \rangle \} \{ \langle false \text{ code} \rangle \}$

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

New: 2012-03-03

11 Setting skip variables

`\skip_add:Nn`
`\skip_add:cn`
`\skip_gadd:Nn`
`\skip_gadd:cn`

`\skip_add:Nn` $\langle skip \rangle$ $\{ \langle skip \text{ expression} \rangle \}$

Adds the result of the $\langle skip \text{ expression} \rangle$ to the current content of the $\langle skip \rangle$.

Updated: 2011-10-22

`\skip_set:Nn`
`\skip_set:cn`
`\skip_gset:Nn`
`\skip_gset:cn`

`\skip_set:Nn` $\langle skip \rangle$ $\{ \langle skip \text{ expression} \rangle \}$

Sets $\langle skip \rangle$ to the value of $\langle skip \text{ expression} \rangle$, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).

Updated: 2011-10-22

`\skip_set_eq:NN`
`\skip_set_eq:(cN|Nc|cc)`
`\skip_gset_eq:NN`
`\skip_gset_eq:(cN|Nc|cc)`

`\skip_set_eq:NN` $\langle skip_1 \rangle$ $\langle skip_2 \rangle$

Sets the content of $\langle skip_1 \rangle$ equal to that of $\langle skip_2 \rangle$.

<code>\skip_sub:Nn</code>	<code>\skip_sub:Nn <skip> {<skip expression>}</code>
<code>\skip_sub:cn</code>	
<code>\skip_gsub:Nn</code>	Subtracts the result of the <i><skip expression></i> from the current content of the <i><skip></i> .
<code>\skip_gsub:cn</code>	

Updated: 2011-10-22

12 Skip expression conditionals

<code>\skip_if_eq_p:n</code> *	<code>\skip_if_eq_p:n {<skipexpr₁>} {<skipexpr₂>}</code>
<code>\skip_if_eq:nnTF</code> *	<code>\skip_if_eq:nnTF</code>
	<code>{<skipexpr₁>} {<skipexpr₂>}</code>
	<code>{<>true code>} {<>false code>}</code>

This function first evaluates each of the *<skip expressions>* 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 {<skipexpr>}</code>
<code>\skip_if_finite:nnTF</code> *	<code>\skip_if_finite:nnTF {<skipexpr>} {<>true code>} {<>false code>}</code>

New: 2012-03-05

Evaluates the *<skip expression>* as described for `\skip_eval:n`, and then tests if all of its components are finite.

13 Using skip expressions and variables

<code>\skip_eval:n</code> *	<code>\skip_eval:n {<skip expression>}</code>
-----------------------------	---

Updated: 2011-10-22

Evaluates the *<skip expression>*, expanding any skips and token list variables within the *<expression>* 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 *<glue denotation>* 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 *<internal glue>*.

<code>\skip_use:N</code> *	<code>\skip_use:N <skip></code>
<code>\skip_use:c</code> *	

Recovers the content of a *<skip>* 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 *<dimension>* is required (such as in the argument of `\skip_eval:n`).

\TeX hackers note: `\skip_use:N` is the \TeX primitive `\the`: this is one of several \LaTeX 3 names for this primitive.

14 Viewing skip variables

<code>\skip_show:N</code>	<code>\skip_show:N <skip></code>
<code>\skip_show:c</code>	

Updated: 2015-08-03

Displays the value of the *<skip>* on the terminal.

`\skip_show:n` `\skip_show:n {⟨skip expression⟩}`
New: 2011-11-22 Displays the result of evaluating the *⟨skip expression⟩* on the terminal.
Updated: 2015-08-07

`\skip_log:N` `\skip_log:N ⟨skip⟩`
`\skip_log:c` Writes the value of the *⟨skip⟩* in the log file.
New: 2014-08-22
Updated: 2015-08-03

`\skip_log:n` `\skip_log:n {⟨skip expression⟩}`
New: 2014-08-22 Writes the result of evaluating the *⟨skip expression⟩* in the log file.
Updated: 2015-08-07

15 Constant skips

`\c_max_skip` The maximum value that can be stored as a skip (equal to `\c_max_dim` in length), with no stretch nor shrink component.
Updated: 2012-11-02

`\c_zero_skip` A zero length as a skip, with no stretch nor shrink component.
Updated: 2012-11-01

16 Scratch skips

`\l_tmpa_skip` Scratch skip 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.
`\l_tmpb_skip`

`\g_tmpa_skip` Scratch skip 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.
`\g_tmpb_skip`

17 Inserting skips into the output

`\skip_horizontal:N` `\skip_horizontal:N ⟨skip⟩`
`\skip_horizontal:c` `\skip_horizontal:n {⟨skipexpr⟩}`
`\skip_horizontal:n` Inserts a horizontal *⟨skip⟩* into the current list.
Updated: 2011-10-22

T_EXhackers note: `\skip_horizontal:N` is the T_EX primitive `\hskip` renamed.

`\skip_vertical:N`
`\skip_vertical:c`
`\skip_vertical:n`

`\skip_vertical:N` $\langle skip \rangle$
`\skip_vertical:n` $\{ \langle skipexpr \rangle \}$

Inserts a vertical $\langle skip \rangle$ into the current list.

Updated: 2011-10-22

T_EXhackers note: `\skip_vertical:N` is the T_EX primitive `\vskip` renamed.

18 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

19 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

<code>\muskip_set:Nn</code>	<code>\muskip_set:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_set:cn</code>	
<code>\muskip_gset:Nn</code>	Sets <i><muskip></i> to the value of <i><muskip expression></i> , which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).
<code>\muskip_gset:cn</code>	

Updated: 2011-10-22

<code>\muskip_set_eq:NN</code>	<code>\muskip_set_eq:NN <muskip₁> <muskip₂></code>
<code>\muskip_set_eq:(cN Nc cc)</code>	
<code>\muskip_gset_eq:NN</code>	Sets the content of <i><muskip₁></i> equal to that of <i><muskip₂></i> .
<code>\muskip_gset_eq:(cN Nc cc)</code>	

<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>	

Updated: 2011-10-22

20 Using muskip expressions and variables

<code>\muskip_eval:n</code> *	<code>\muskip_eval:n {<muskip expression>}</code>
-------------------------------	---

Updated: 2011-10-22

Evaluates the *<muskip expression>*, expanding any skips and token list variables within the *<expression>* to their content (without requiring `\muskip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a *<mu glue denotation>* after two expansions. This is expressed in `mu`, and requires suitable termination if used in a T_EX-style assignment as it is *not* an *<internal mu glue>*.

<code>\muskip_use:N</code> *	<code>\muskip_use:N <muskip></code>
<code>\muskip_use:c</code> *	

Recovers the content of a *<skip>* 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 *<dimension>* is required (such as in the argument of `\muskip_eval:n`).

T_EXhackers note: `\muskip_use:N` is the T_EX primitive `\the`: this is one of several L^AT_EX3 names for this primitive.

21 Viewing muskip variables

<code>\muskip_show:N</code>	<code>\muskip_show:N <muskip></code>
<code>\muskip_show:c</code>	
	Displays the value of the <i><muskip></i> on the terminal.

Updated: 2015-08-03

<code>\muskip_show:n</code>	<code>\muskip_show:n {<muskip expression>}</code>
-----------------------------	---

New: 2011-11-22
Updated: 2015-08-07

Displays the result of evaluating the *<muskip expression>* on the terminal.

`\muskip_log:N` `\muskip_log:N` $\langle muskip \rangle$
`\muskip_log:c` Writes the value of the $\langle muskip \rangle$ in the log file.
New: 2014-08-22
Updated: 2015-08-03

`\muskip_log:n` `\muskip_log:n` $\{\langle muskip\ expression \rangle\}$
New: 2014-08-22 Writes the result of evaluating the $\langle muskip\ expression \rangle$ in the log file.
Updated: 2015-08-07

22 Constant muskips

`\c_max_muskip` The maximum value that can be stored as a muskip, with no stretch nor shrink component.

`\c_zero_muskip` A zero length as a muskip, with no stretch nor shrink component.

23 Scratch muskips

`\l_tmpa_muskip` Scratch muskip for local assignment. These are never used by the kernel code, and so
`\l_tmpb_muskip` 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_muskip` Scratch muskip for global assignment. These are never used by the kernel code, and so
`\g_tmpb_muskip` 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.

24 Primitive conditional

`\if_dim:w` `\if_dim:w` $\langle dimen_1 \rangle$ $\langle relation \rangle$ $\langle dimen_2 \rangle$
 $\langle true\ code \rangle$
 `\else:`
 $\langle false \rangle$
 `\fi:`
Compare two dimensions. The $\langle relation \rangle$ is one of $<$, $=$ or $>$ with category code 12.

T_EXhackers note: This is the T_EX primitive `\ifdim`.

25 Internal functions

<code>_dim_eval:w</code>	*	<code>_dim_eval:w</code> $\langle dimexpr \rangle$ <code>_dim_eval_end:</code>
<code>_dim_eval_end:</code>	*	Evaluates $\langle dimension\ expression \rangle$ as described for <code>\dim_eval:n</code> . The evaluation stops when an unexpandable token which is not a valid part of a dimension is read or when <code>_dim_eval_end:</code> is reached. The latter is gobbled by the scanner mechanism: <code>_dim_eval_end:</code> itself is unexpandable but used correctly the entire construct is expandable.

TeXhackers note: This is the ε -TeX primitive `\dimexpr`.

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: 2015-11-07

```
\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
```

$\langle key \rangle$.bool_set:N = $\langle boolean \rangle$

```
.bool_set:c
```

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_gset:N
```

```
.bool_gset:c
```

Updated: 2013-07-08

```
.bool_set_inverse:N
```

$\langle key \rangle$.bool_set_inverse:N = $\langle boolean \rangle$

```
.bool_set_inverse:c
```

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.

```
.bool_gset_inverse:N
```

```
.bool_gset_inverse:c
```

New: 2011-08-28
Updated: 2013-07-08

```
.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
```

$\langle key \rangle$.choices:nn = $\{\langle choices \rangle\} \{\langle code \rangle\}$

```
.choices:(Vn|on|xn)
```

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.

New: 2011-08-21
Updated: 2013-07-10

```
.clist_set:N
```

$\langle key \rangle$.clist_set:N = $\langle comma list variable \rangle$

```
.clist_set:c
```

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.

```
.clist_gset:N
```

```
.clist_gset:c
```

New: 2011-09-11

```
.code:n
```

$\langle key \rangle$.code:n = $\{\langle code \rangle\}$

Updated: 2013-07-10

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 }
```

`.initial:n`
`.initial:(V|o|x)`
Updated: 2013-07-09

`<key> .initial:n = {<value>}`
Initialises the `<key>` with the `<value>`, equivalent to
`\keys_set:nn {<module>} { <key> = <value> }`

`.int_set:N`
`.int_set:c`
`.int_gset:N`
`.int_gset:c`

`<key> .int_set:N = <integer>`
Defines `<key>` to set `<integer>` to `<value>` (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.

`.meta:n`
Updated: 2013-07-10

`<key> .meta:n = {<keyval list>}`
Makes `<key>` a meta-key, which will set `<keyval list>` in one go. If `<key>` is given with a value at the time the key is used, then the value will be passed through to the subsidiary `<keys>` for processing (as #1).

`.meta:nn`
New: 2013-07-10

`<key> .meta:nn = {<path>} {<keyval list>}`
Makes `<key>` a meta-key, which will set `<keyval list>` in one go using the `<path>` in place of the current one. If `<key>` is given with a value at the time the key is used, then the value will be passed through to the subsidiary `<keys>` for processing (as #1).

`.multichoice:`
New: 2011-08-21

`<key> .multichoice:`
Sets `<key>` to act as a multiple choice key. Each valid choice for `<key>` must then be created, as discussed in section 3.

`.multichoices:nn`
`.multichoices:(Vn|on|xn)`
New: 2011-08-21
Updated: 2013-07-10

`<key> .multichoices:nn {<choices>} {<code>}`
Sets `<key>` to act as a multiple choice key, and defines a series `<choices>` which are implemented using the `<code>`. Inside `<code>`, `\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 `<choices>` (indexed from 1). Choices are discussed in detail in section 3.

`.skip_set:N`
`.skip_set:c`
`.skip_gset:N`
`.skip_gset:c`

`<key> .skip_set:N = <skip>`
Defines `<key>` to set `<skip>` to `<value>` (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.

`.tl_set:N`
`.tl_set:c`
`.tl_gset:N`
`.tl_gset:c`

`<key> .tl_set:N = <token list variable>`
Defines `<key>` to set `<token list variable>` to `<value>`. If the variable does not exist, it is created globally at the point that the key is set up.

`.tl_set_x:N`
`.tl_set_x:c`
`.tl_gset_x:N`
`.tl_gset_x:c`

`<key> .tl_set_x:N = <token list variable>`
Defines `<key>` to set `<token list variable>` to `<value>`, which will be subjected to an x-type expansion (*i.e.* using `\tl_set:Nx`). If the variable does not exist, it is created globally at the point that the key is set up.

`.undefine:` `<key> .undefine:`

New: 2015-07-14 Removes the definition of the `<key>` within the current scope.

`.value_forbidden:n` `<key> .value_forbidden:n = true|false`

New: 2015-07-14 Specifies that `<key>` cannot receive a `<value>` when used. If a `<value>` is given then an error will be issued. Setting the property `false` cancels the restriction.

`.value_required:n` `<key> .value_required:n = true|false`

New: 2015-07-14 Specifies that `<key>` must receive a `<value>` when used. If a `<value>` is not given then an error will be issued. Setting the property `false` 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: 2015-11-07

`\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.

```

\keys_set_filter:nnnN      \keys_set_filter:nnnN {<module>} {<groups>} {<keyval list>} <tl>
\keys_set_filter:(nnVN|nnvN|nnoN)
\keys_set_filter:nnn
\keys_set_filter:(nnV|nnv|nno)

```

New: 2013-07-14
Updated: 2017-05-27

Activates key filtering in an “opt-out” sense: keys assigned to any of the *<groups>* specified are ignored. The *<groups>* 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 *<tl>* in a comma-separated form (*i.e.* an edited version of the *<keyval list>*). The `\keys_set_filter:nnn` version skips this stage.

Use of `\keys_set_filter:nnnN` can be nested, with the correct residual *<keyval list>* returned at each stage.

```

\keys_set_groups:nnn      \keys_set_groups:nnn {<module>} {<groups>} {<keyval list>}
\keys_set_groups:(nnV|nnv|nno)

```

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 *<groups>* specified are set. The *<groups>* 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

```

\keys_if_exist_p:nn *      \keys_if_exist_p:nn {<module>} {<key>}
\keys_if_exist:nnTF *      \keys_if_exist:nnTF {<module>} {<key>} {<true code>} {<false code>}

```

Updated: 2015-11-07 Tests if the *<key>* exists for *<module>*, *i.e.* if any code has been defined for *<key>*.

```

\keys_if_choice_exist_p:nnn * \keys_if_choice_exist_p:nnn {<module>} {<key>} {<choice>}
\keys_if_choice_exist:nnnTF * \keys_if_choice_exist:nnnTF {<module>} {<key>} {<choice>} {<true code>}
                               {<false code>}

```

New: 2011-08-21
Updated: 2015-11-07

Tests if the *<choice>* is defined for the *<key>* within the *<module>*, *i.e.* if any code has been defined for *<key>/<choice>*. The test is **false** if the *<key>* itself is not defined.

```

\keys_show:nn      \keys_show:nn {<module>} {<key>}

```

Updated: 2015-08-09

Displays in the terminal the information associated to the *<key>* for a *<module>*, including the function which is used to actually implement it.

```

\keys_log:nn      \keys_log:nn {<module>} {<key>}

```

New: 2014-08-22
Updated: 2015-08-09

Writes in the log file the information associated to the *<key>* for a *<module>*. 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 <function1> <function2> {<key-value list>}`

Parses the *<key-value list>* into a series of *<keys>* and associated *<values>*, or keys alone (if no *<value>* was given). *<function₁>* should take one argument, while *<function₂>* should absorb two arguments. After `\keyval_parse:NNn` has parsed the *<key-value list>*, *<function₁>* is used to process keys given with no value and *<function₂>* is used to process keys given with a value. The order of the *<keys>* in the *<key-value list>* 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 *<key>* and *<value>*, then one *outer* set of braces is removed from the *<key>* and *<value>* 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: $x < y$, $x <= y$, $x >? y$, $x != y$ 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 pdfTeX and LuaTeX engines.
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
- Automatic conversion (no need for `\langle type \rangle_use:N`) of integer, dimension, and skip variables to floating points, expressing dimensions in points and ignoring the stretch and shrink components of skips.

Floating point numbers can be given either explicitly (in a form such as `1.234e-34`, or `-.0001`), or as a stored floating point variable, which is automatically replaced by its current value. 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} $.`

But in all fairness, this 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 { \calcnun } { m }
{ \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnun { 2 pi * sin ( 2.3 ^ 5 ) }
\end{document}
```

1 Creating and initialising floating point variables

<code>\fp_new:N</code>	<code>\fp_new:N <fp var></code>
<code>\fp_new:c</code>	Creates a new <i><fp var></i> or raises an error if the name is already taken. The declaration is global. The <i><fp var></i> is initially +0.

Updated: 2012-05-08

<code>\fp_const:Nn</code>	<code>\fp_const:Nn <fp var> {<floating point expression>}</code>
<code>\fp_const:cn</code>	Creates a new constant <i><fp var></i> or raises an error if the name is already taken. The <i><fp var></i> is set globally equal to the result of evaluating the <i><floating point expression></i> .

Updated: 2012-05-08

<code>\fp_zero:N</code>	<code>\fp_zero:N <fp var></code>
<code>\fp_zero:c</code>	Sets the <i><fp var></i> to +0.
<code>\fp_gzero:N</code>	
<code>\fp_gzero:c</code>	

Updated: 2012-05-08

<code>\fp_zero_new:N</code>	<code>\fp_zero_new:N <fp var></code>
<code>\fp_zero_new:c</code>	Ensures that the <i><fp var></i> exists globally by applying <code>\fp_new:N</code> if necessary, then applies <code>\fp_(g)zero:N</code> to leave the <i><fp var></i> set to +0.
<code>\fp_gzero_new:N</code>	
<code>\fp_gzero_new:c</code>	

Updated: 2012-05-08

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>	Sets <i><fp var></i> equal to the result of computing the <i><floating point expression></i> .
<code>\fp_gset:Nn</code>	
<code>\fp_gset:cn</code>	

Updated: 2012-05-08

<code>\fp_set_eq:NN</code>	<code>\fp_set_eq:NN</code> $\langle fp\ var_1 \rangle$ $\langle fp\ var_2 \rangle$
<code>\fp_set_eq:(cN Nc cc)</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:NN</code>	
<code>\fp_gset_eq:(cN Nc cc)</code>	

Updated: 2012-05-08

<code>\fp_add:Nn</code>	<code>\fp_add:Nn</code> $\langle fp\ var \rangle$ $\{ \langle floating\ point\ expression \rangle \}$
<code>\fp_add:cn</code>	
<code>\fp_gadd:Nn</code>	Adds the result of computing the $\langle floating\ point\ expression \rangle$ to the $\langle fp\ var \rangle$.
<code>\fp_gadd:cn</code>	

Updated: 2012-05-08

<code>\fp_sub:Nn</code>	<code>\fp_sub:Nn</code> $\langle fp\ var \rangle$ $\{ \langle floating\ point\ expression \rangle \}$
<code>\fp_sub:cn</code>	
<code>\fp_gsub:Nn</code>	Subtracts the result of computing the $\langle floating\ point\ expression \rangle$ from the $\langle fp\ var \rangle$.
<code>\fp_gsub:cn</code>	

Updated: 2012-05-08

3 Using floating point numbers

<code>\fp_eval:n</code> *	<code>\fp_eval:n</code> $\{ \langle floating\ point\ expression \rangle \}$
New: 2012-05-08	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. This function is identical to <code>\fp_to_decimal:n</code> .
Updated: 2012-07-08	

<code>\fp_to_decimal:N</code> *	<code>\fp_to_decimal:N</code> $\langle fp\ var \rangle$
<code>\fp_to_decimal:c</code> *	<code>\fp_to_decimal:n</code> $\{ \langle floating\ point\ expression \rangle \}$
<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.
New: 2012-05-08	
Updated: 2012-07-08	

<code>\fp_to_dim:N</code> *	<code>\fp_to_dim:N</code> $\langle fp\ var \rangle$
<code>\fp_to_dim:c</code> *	<code>\fp_to_dim:n</code> $\{ \langle floating\ point\ expression \rangle \}$
<code>\fp_to_dim:n</code> *	Evaluates the $\langle floating\ point\ expression \rangle$ 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. 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</code>	$\langle fp \text{ var} \rangle$
<code>\fp_to_int:c</code>	★	<code>\fp_to_int:n</code>	$\{\langle floating \text{ point expression} \rangle\}$
<code>\fp_to_int:n</code>	★		Evaluates the $\langle floating \text{ point expression} \rangle$, 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. 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</code>	$\langle fp \text{ var} \rangle$
<code>\fp_to_scientific:c</code>	★	<code>\fp_to_scientific:n</code>	$\{\langle floating \text{ point expression} \rangle\}$
<code>\fp_to_scientific:n</code>	★		Evaluates the $\langle floating \text{ point expression} \rangle$ and expresses the result in scientific notation:

New: 2012-05-08
Updated: 2016-03-22

$\langle optional - \rangle \langle digit \rangle . \langle 15 \text{ digits} \rangle e \langle optional \text{ sign} \rangle \langle exponent \rangle$

The leading $\langle digit \rangle$ 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).

<code>\fp_to_tl:N</code>	★	<code>\fp_to_tl:N</code>	$\langle fp \text{ var} \rangle$
<code>\fp_to_tl:c</code>	★	<code>\fp_to_tl:n</code>	$\{\langle floating \text{ point expression} \rangle\}$
<code>\fp_to_tl:n</code>	★		Evaluates the $\langle floating \text{ point expression} \rangle$ 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.

Updated: 2016-03-22

<code>\fp_use:N</code>	★	<code>\fp_use:N</code>	$\langle fp \text{ var} \rangle$
<code>\fp_use:c</code>	★		Inserts the value of the $\langle fp \text{ var} \rangle$ 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. 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</code>	$\langle fp \text{ var} \rangle$
<code>\fp_if_exist_p:c</code>	★	<code>\fp_if_exist:N</code>	$\langle fp \text{ var} \rangle$ $\{\langle true \text{ code} \rangle\}$ $\{\langle false \text{ code} \rangle\}$
<code>\fp_if_exist:N</code>	★		Tests whether the $\langle fp \text{ var} \rangle$ is currently defined. This does not check that the $\langle fp \text{ var} \rangle$ really is a floating point variable.

Updated: 2012-05-08

```

\fp_compare_p:nNn * \fp_compare_p:nNn {<fpexpr1>} <relation> {<fpexpr2>}
\fp_compare:nNnTF * \fp_compare:nNnTF {<fpexpr1>} <relation> {<fpexpr2>} {<>true code>} {<>false code>}

```

Updated: 2012-05-08

Compares the $\langle fpexpr_1 \rangle$ and the $\langle fpexpr_2 \rangle$, and returns **true** if the $\langle relation \rangle$ is obeyed. Two floating point numbers x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or x and y are not ordered. The latter case occurs exactly when one or both operands is NaN, and this relation is denoted by the symbol ?. 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

```

```

\fp_compare_p:n * \fp_compare_p:n
\fp_compare:nTF * {
  <fpexpr1> <relation1>
  ...
  <fpexprN> <relationN>
  <fpexprN+1>
}
\fp_compare:nTF
{
  <fpexpr1> <relation1>
  ...
  <fpexprN> <relationN>
  <fpexprN+1>
}
{<>true code>} {<>false code>}

```

Updated: 2012-12-14

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 point numbers x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or x and y are not ordered. The latter case occurs exactly when one or both operands is NaN, and this relation is denoted by the symbol ?. 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 floatin 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 {<fpexpr1>} <relation> {<fpexpr2>} {<code>}</code>
New: 2012-08-16	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:nNnTF</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 .
<code>\fp_do_while:nNnn</code> ☆	<code>\fp_do_while:nNnn {<fpexpr1>} <relation> {<fpexpr2>} {<code>}</code>
New: 2012-08-16	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:nNnTF</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 .
<code>\fp_until_do:nNnn</code> ☆	<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 .
<code>\fp_while_do:nNnn</code> ☆	<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 .
<code>\fp_do_until:nn</code> ☆	<code>\fp_do_until:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16	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 .
<code>\fp_do_while:nn</code> ☆	<code>\fp_do_while:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16	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 .
<code>\fp_until_do:nn</code> ☆	<code>\fp_until_do:nn { <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: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 .

`\fp_while_do:nn` ☆

New: 2012-08-16

`\fp_while_do:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}`

Evaluates the relationship between the two *<floating point expressions>* as described for `\fp_compare:nTF`, and then places the *<code>* in the input stream if the *<relation>* is true. After the *<code>* has been processed by \TeX 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>*, all of which should be floating point expressions. 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]
```

\TeX hackers 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. 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. 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`

One as an fp: useful for comparisons in some places.

New: 2012-05-08

`\c_inf_fp`
`\c_minus_inf_fp`

Infinity, with either sign. These can be input directly in a floating point expression as `inf` and `-inf`.

New: 2012-05-08

`\c_e_fp`

The value of the base of the natural logarithm, $e = \exp(1)$.

Updated: 2012-05-08

`\c_pi_fp`

The value of π . This can be input directly in a floating point expression as `pi`.

Updated: 2013-11-17

`\c_one_degree_fp`

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 `deg`.

New: 2012-05-08

Updated: 2013-11-17

`\l_tmpa_fp`

`\l_tmpb_fp`

Scratch floating points 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_fp`

`\g_tmpb_fp`

Scratch floating points 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.

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 $\sin(\infty)$, 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:n`) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

`\fp_trap:n`
New: 2012-07-19
Updated: 2017-02-13

`\fp_trap:n` $\langle exception \rangle$ $\langle trap\ type \rangle$
All occurrences of the $\langle exception \rangle$ (`overflow`, `underflow`, `invalid_operation` or `division_by_zero`) within the current group are treated as $\langle trap\ type \rangle$, which can be

- **none**: the $\langle exception \rangle$ will be entirely ignored, and leave no trace;
- **flag**: the $\langle exception \rangle$ will turn the corresponding flag on when it occurs;
- **error**: additionally, the $\langle exception \rangle$ 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

`\fp_show:N`
`\fp_show:c`
`\fp_show:n`
New: 2012-05-08
Updated: 2015-08-07

`\fp_show:N` $\langle fp\ var \rangle$
`\fp_show:n` $\langle floating\ point\ expression \rangle$
Evaluates the $\langle floating\ point\ expression \rangle$ and displays the result in the terminal.

`\fp_log:N`
`\fp_log:c`
`\fp_log:n`
New: 2014-08-22
Updated: 2015-08-07

`\fp_log:N` $\langle fp\ var \rangle$
`\fp_log:n` $\langle floating\ point\ expression \rangle$
Evaluates the $\langle floating\ point\ expression \rangle$ and writes the result in the log file.

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.

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).

The precedence of operations can be overridden using parentheses. In particular, those precedences imply that

$$\begin{aligned} \text{sin2pi} &= \sin(2\pi) = 0, \\ 2^{2\max(3,4)} &= 2^{2\max(3,4)} = 256. \end{aligned}$$

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`.

```
?: \fp_eval:n { <operand1> ? <operand2> : <operand3> }
```

The ternary operator `?:` results in `<operand2>` if `<operand1>` is true, and `<operand3>` if it is false (equal to ± 0). All three `<operands>` are evaluated in all cases. 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 `<operand1>` is true (non-zero), use that value, otherwise the value of `<operand2>`. Both `<operands>` are evaluated in all cases.

`&&` `\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.

`<` `\fp_eval:n`
`=` `{`
`>` `<operand1> <relation1>`
`?` `...`
`<operandN> <relationN>`
`<operandN+1>`
`}`

Updated: 2013-12-14

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 $\langle operand_i \rangle \langle relation_j \rangle \langle operand_{i+1} \rangle$ are true, and `+0` otherwise. All $\langle operands \rangle$ are evaluated 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.

`*` `\fp_eval:n { <operand1> * <operand2> }`
`/` `\fp_eval:n { <operand1> / <operand2> }`

Computes the product or the ratio of its two $\langle operands \rangle$. 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.

`+` `\fp_eval:n { + <operand> }`
`-` `\fp_eval:n { - <operand> }`
`!` `\fp_eval:n { ! <operand> }`

The unary `+` does nothing, the unary `-` changes the sign of the $\langle operand \rangle$, and `!` $\langle operand \rangle$ evaluates to 1 if $\langle operand \rangle$ is false 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 $\langle operand_1 \rangle$ to the power $\langle operand_2 \rangle$. This operation is right associative, hence `2 ** 2 ** 3` equals $2^{2^3} = 256$. If $\langle operand_1 \rangle$ is negative or -0 then: the result’s sign is `+` if the $\langle operand_2 \rangle$ is infinite and $(-1)^p$ if the $\langle operand_2 \rangle$ is $p/5^q$ with p, q integers; the result is `+0` if `abs(<operand1>)^<operand2>` 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.

`abs` `\fp_eval:n { abs(<fpexpr>) }`

Computes the absolute value of the $\langle fpexpr \rangle$. This function does not raise any exception beyond those raised when computing its operand $\langle fpexpr \rangle$. See also `\fp_abs:n`.

exp `\fp_eval:n { exp(<fpexpr>) }`
 Computes the exponential of the $\langle fpexpr \rangle$. “Underflow” and “overflow” occur when appropriate.

ln `\fp_eval:n { ln(<fpexpr>) }`
 Computes the natural logarithm of the $\langle fpexpr \rangle$. 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.

max `\fp_eval:n { max(<fpexpr1> , <fpexpr2> , ...) }`
min `\fp_eval:n { min(<fpexpr1> , <fpexpr2> , ...) }`
 Evaluates each $\langle fpexpr \rangle$ and computes the largest (smallest) of those. If any of the $\langle fpexpr \rangle$ is a NaN, the result is NaN. Those operations do not raise exceptions.

round `\fp_eval:n { round (<fpexpr>) }`
trunc `\fp_eval:n { round (<fpexpr1> , <fpexpr2>) }`
ceil `\fp_eval:n { round (<fpexpr1> , <fpexpr2> , <fpexpr3>) }`
floor `\fp_eval:n { round (<fpexpr1> , <fpexpr2> , <fpexpr3>) }`
 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.

New: 2013-12-14
 Updated: 2015-08-08

- **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 $\langle fpexpr_2 \rangle < -9984$).

sign `\fp_eval:n { sign(<fpexpr>) }`
 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. This operation does not raise exceptions.

```

sin      \fp_eval:n { sin( <fpexpr> ) }
cos      \fp_eval:n { cos( <fpexpr> ) }
tan      \fp_eval:n { tan( <fpexpr> ) }
cot      \fp_eval:n { cot( <fpexpr> ) }
csc      \fp_eval:n { csc( <fpexpr> ) }
sec      \fp_eval:n { sec( <fpexpr> ) }

```

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(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.

```

sind     \fp_eval:n { sind( <fpexpr> ) }
cosd     \fp_eval:n { cosd( <fpexpr> ) }
tand     \fp_eval:n { tand( <fpexpr> ) }
cotd     \fp_eval:n { cotd( <fpexpr> ) }
cscd     \fp_eval:n { cscd( <fpexpr> ) }
secd     \fp_eval:n { secd( <fpexpr> ) }

```

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.

```

asin     \fp_eval:n { asin( <fpexpr> ) }
acos     \fp_eval:n { acos( <fpexpr> ) }
acsc     \fp_eval:n { acsc( <fpexpr> ) }
asec     \fp_eval:n { asec( <fpexpr> ) }

```

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.

```

asind    \fp_eval:n { asind( <fpexpr> ) }
acosd    \fp_eval:n { acosd( <fpexpr> ) }
acscd    \fp_eval:n { acscd( <fpexpr> ) }
asecd    \fp_eval:n { asecd( <fpexpr> ) }

```

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.

<code>atan</code>	<code>\fp_eval:n { atan(<fpexpr>) }</code>
<code>acot</code>	<code>\fp_eval:n { atan(<fpexpr1> , <fpexpr2>) }</code>

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 `<fpexpr>`: 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. Only the “underflow” exception can occur.

<code>atand</code>	<code>\fp_eval:n { atand(<fpexpr>) }</code>
<code>acotd</code>	<code>\fp_eval:n { atand(<fpexpr1> , <fpexpr2>) }</code>

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 `<fpexpr>`: 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. Only the “underflow” exception can occur.

<code>sqrt</code>	<code>\fp_eval:n { sqrt(<fpexpr>) }</code>
-------------------	--

New: 2013-12-14 Computes the square root of the `<fpexpr>`. The “invalid operation” is raised when the `<fpexpr>` is negative; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\text{NaN}} = \text{NaN}$.

rand `\fp_eval:n { rand() }`

New: 2016-12-05

 Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. Available in pdfTeX and LuaTeX engines only.

TeXhackers note: This is based on pseudo-random numbers provided by the engine’s primitive `\pdfuniformdeviate` in pdfTeX and `\uniformdeviate` in LuaTeX. The underlying code in pdfTeX and LuaTeX 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.

The random seed can be queried using `\pdfrandomseed` and set using `\pdfsetrandomseed` (in LuaTeX `\randomseed` and `\setrandomseed`). 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 $[0, 2^{28} - 1]$.

randint `\fp_eval:n { randint(<fpexpr>) }`

New: 2016-12-05

`\fp_eval:n { randint(<fpexpr12
 Produces a pseudo-random integer between 1 and <fpexpr> or between <fpexpr1 and <fpexpr2 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.`

inf `\c_inf_fp`, `\c_minus_inf_fp` and `\c_nan_fp`.
nan

pi The value of π (see `\c_pi_fp`).

deg The value of 1° in radians (see `\c_one_degree_fp`).

—	
em	Those units of measurement are equal to their values in pt, namely
ex	
in	1in = 72.27pt
pt	1pt = 1pt
pc	1pc = 12pt
cm	
mm	1cm = $\frac{1}{2.54}$ in = 28.45275590551181pt
dd	
cc	1mm = $\frac{1}{25.4}$ in = 2.845275590551181pt
nd	
nc	1dd = 0.376065mm = 1.07000856496063pt
bp	1cc = 12dd = 12.84010277952756pt
sp	1nd = 0.375mm = 1.066978346456693pt
—	1nc = 12nd = 12.80374015748031pt
	1bp = $\frac{1}{72}$ in = 1.00375pt
	1sp = 2 ⁻¹⁶ pt = 1.52587890625e - 5pt.

The values of the (font-dependent) units `em` and `ex` are gathered from \TeX when the surrounding floating point expression is evaluated.

—	
true	Other names for 1 and +0.
false	
—	

—	
<code>\fp_abs:n</code> ★	<code>\fp_abs:n</code> { <i>floating point expression</i> }
New: 2012-05-14 Updated: 2012-07-08	Evaluates the <i>floating point expression</i> as described for <code>\fp_eval:n</code> and leaves the absolute value of the result in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, <code>abs()</code> can be used.
—	

—	
<code>\fp_max:nn</code> ★	<code>\fp_max:nn</code> { <i>fp expression 1</i> } { <i>fp expression 2</i> }
<code>\fp_min:nn</code> ★	Evaluates the <i>floating point expressions</i> as described for <code>\fp_eval:n</code> and leaves the resulting larger (<code>max</code>) or smaller (<code>min</code>) value in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, <code>max()</code> and <code>min()</code> can be used.
New: 2012-09-26	
—	

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.

- Decide what exponent range to consider.
- Support signalling `nan`.

- Modulo and remainder, and rounding functions `quantize`, `quantize0`, `quantize+`, `quantize-`, `quantize=`, `round=`. Should the modulo also be provided as (catcode 12) `%`?
- `\fp_format:nn` $\{\langle fpexpr \rangle\}$ $\{\langle format \rangle\}$, but what should $\langle format \rangle$ 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.

- 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 TeX fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)?

Part XXII

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.

Part XXIII

The `l3tl-build` package: building token lists

1 `l3tl-build` documentation

This module provides no user function: it is meant for kernel use only.

There are two main ways of building token lists from individual tokens. Either in one go within an `x`-expanding assignment, or by repeatedly using `\tl_put_right:Nn`. The first method takes a linear time, but only allows expandable operations. The second method takes a time quadratic in the length of the token list, but allows expandable and non-expandable operations.

The goal of this module is to provide functions to build a token list piece by piece in linear time, while allowing non-expandable operations. This is achieved by abusing `\toks`: adding some tokens to the token list is done by storing them in a free token register (time $O(1)$ for each such operation). Those token registers are only put together at the end, within an `x`-expanding assignment, which takes a linear time.⁵ Of course, all this must be done in a group: we can't go and clobber the values of legitimate `\toks` used by L^AT_EX 2_ε.

Since none of the current applications need the ability to insert material on the left of the token list, I have not implemented that. This could be done for instance by using odd-numbered `\toks` for the left part, and even-numbered `\toks` for the right part.

1.1 Internal functions

<code>__tl_build:Nw</code>	<code>__tl_build:Nw <tl var> ...</code>
<code>__tl_gbuild:Nw</code>	<code>__tl_build_one:n {<tokens1>} ...</code>
<code>__tl_build_x:Nw</code>	<code>__tl_build_one:n {<tokens2>} ...</code>
<code>__tl_gbuild_x:Nw</code>	<code>...</code>

`__tl_build_end:`

Defines the `<tl var>` to contain the contents of `<tokens1>` followed by `<tokens2>`, etc. This is built in such a way to be more efficient than repeatedly using `\tl_put_right:Nn`. The code in “...” does not need to be expandable. The commands `__tl_build:Nw` and `__tl_build_end:` start and end a group. The assignment to the `<tl var>` occurs just after the end of that group, using `\tl_set:Nn`, `\tl_gset:Nn`, `\tl_set:Nx`, or `\tl_gset:Nx`.

<code>__tl_build_one:n</code>	<code>__tl_build_one:n {<tokens>}</code>
--------------------------------	---

`__tl_build_one:(o|x)`

This function may only be used within the scope of a `__tl_build:Nw` function. It adds the `<tokens>` on the right of the current token list.

`__tl_build_end:`

Ends the scope started by `__tl_build:Nw`, and performs the relevant assignment.

⁵If we run out of token registers, then the currently filled-up `\toks` are put together in a temporary token list, and cleared, and we ultimately use `\tl_put_right:Nx` to put those chunks together. Hence the true asymptotic is quadratic, with a very small constant.

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_show_analysis:N` `\tl_show_analysis:n` $\langle token\ list \rangle$

`\tl_show_analysis:n`

New: 2017-05-26

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.

Part XXV

The `l3regex` package: regular expressions in `TeX`

1 Regular expressions

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.

1.1 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 `TEX` 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).
- `[+-_]*(\\d+|\\c.)_*` 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.*, alpha-numeric and underscore, equivalent to `[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 } \1_tmpa_int` yields 2, but replacing **\G** by **^** would result in `\1_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 `\t1_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.

1.2 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.`

1.3 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`

`\regex_new:N` $\langle regex\ var \rangle$

New: 2017-05-26

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_set:Nn` $\langle regex\ var \rangle$ $\{ \langle regex \rangle \}$

`\regex_gset:Nn`

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_const:Nn`

New: 2017-05-26

```
\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` $\{ \langle regex \rangle \}$

`\regex_show:N`

Shows how `l3regex` interprets the $\langle regex \rangle$. For instance, `\regex_show:n { \A X|Y }` shows

New: 2017-05-26

```
+--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.

1.4 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` $\{ \langle regex \rangle \}$ $\{ \langle token\ list \rangle \}$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$

`\regex_match:NnTF`

Tests whether the $\langle regular\ expression \rangle$ matches any part of the $\langle token\ list \rangle$. For instance,

New: 2017-05-26

```
\regex_match:nnTF { b [cde]* } { abecdcx } { 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` $\langle regex \rangle$ $\langle token list \rangle$ $\langle int var \rangle$

Sets $\langle int var \rangle$ within the current TeX group level equal to the number of times $\langle regular expression \rangle$ appears in $\langle token list \rangle$. The search starts by finding the left-most longest match, respecting greedy and ungreedy 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.

1.5 Submatch extraction

`\regex_extract_once:nnNTF`
`\regex_extract_once:NnNTF`

New: 2017-05-26

`\regex_extract_once:nnN` $\langle regex \rangle$ $\langle token list \rangle$ $\langle seq var \rangle$
`\regex_extract_once:nnNTF` $\langle regex \rangle$ $\langle token list \rangle$ $\langle seq var \rangle$ $\langle true code \rangle$ $\langle false code \rangle$

Finds the first match of the $\langle regular expression \rangle$ in the $\langle token list \rangle$. If it exists, the match is stored as the first item of the $\langle seq var \rangle$, and further items are the contents of capturing groups, in the order of their opening parenthesis. The $\langle seq var \rangle$ is assigned locally. If there is no match, the $\langle seq var \rangle$ is cleared. The testing versions insert the $\langle true code \rangle$ into the input stream if a match was found, and the $\langle false code \rangle$ 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:nnNTF`
`\regex_extract_all:NnNTF`

New: 2017-05-26

`\regex_extract_all:nnN` $\langle regex \rangle$ $\langle token list \rangle$ $\langle seq var \rangle$
`\regex_extract_all:nnNTF` $\langle regex \rangle$ $\langle token list \rangle$ $\langle seq var \rangle$ $\langle true code \rangle$ $\langle false code \rangle$

Finds all matches of the $\langle regular expression \rangle$ in the $\langle token list \rangle$, and stores all the submatch information in a single sequence (concatenating the results of multiple `\regex_extract_once:nnN` calls). The $\langle seq var \rangle$ is assigned locally. If there is no match, the $\langle seq var \rangle$ is cleared. The testing versions insert the $\langle true code \rangle$ into the input stream if a match was found, and the $\langle false code \rangle$ 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:nnNTF \regex_split:nnN {<regular expression>} {<token list>} <seq var>
\regex_split:NnNTF \regex_split:nnNTF {<regular expression>} {<token list>} <seq var> {<>true code>}
                                                           {<>false code>}

```

New: 2017-05-26

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.

1.6 Replacement

```

\regex_replace_once:nnNTF \regex_replace_once:nnN {<regular expression>} {<replacement>} <tl var>
\regex_replace_once:NnNTF \regex_replace_once:nnNTF {<regular expression>} {<replacement>} <tl var> {<>true
                                                           code>} {<>false code>}

```

New: 2017-05-26

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:nnNTF \regex_replace_all:nnN {<regular expression>} {<replacement>} <tl var>
\regex_replace_all:NnNTF \regex_replace_all:nnNTF {<regular expression>} {<replacement>} <tl var> {<>true
                                                           code>} {<>false code>}

```

New: 2017-05-26

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>*.

1.7 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_t1` to build `_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 pdf \TeX .
- 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 \TeX ’s own `\^x`.

- Comments: \TeX already has its own system for comments.
- $\backslash Q \dots \backslash E$ escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- $\backslash C$ single byte in UTF-8 mode: $XeTeX$ and $LuaTeX$ 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 <box></code>
<code>\box_new:c</code>	Creates a new <code><box></code> or raises an error if the name is already taken. The declaration is global. The <code><box></code> is initially void.

<code>\box_clear:N</code>	<code>\box_clear:N <box></code>
<code>\box_clear:c</code>	Clears the content of the <code><box></code> 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 <box></code>
<code>\box_clear_new:c</code>	Ensures that the <code><box></code> exists globally by applying <code>\box_new:N</code> if necessary, then applies <code>\box_(g)clear:N</code> to leave the <code><box></code> empty.
<code>\box_gclear_new:N</code>	
<code>\box_gclear_new:c</code>	

<code>\box_set_eq:NN</code>	<code>\box_set_eq:NN <box1> <box2></code>
<code>\box_set_eq:(cN Nc cc)</code>	Sets the content of <code><box1></code> equal to that of <code><box2></code> .
<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 <box1> <box2></code>
<code>\box_set_eq_clear:(cN Nc cc)</code>	Sets the content of <code><box1></code> within the current TeX group equal to that of <code><box2></code> , then clears <code><box2></code> globally.

<code>\box_gset_eq_clear:NN</code>	<code>\box_gset_eq_clear:NN <box1> <box2></code>
<code>\box_gset_eq_clear:(cN Nc cc)</code>	Sets the content of <code><box1></code> equal to that of <code><box2></code> , then clears <code><box2></code> . These assignments are global.

<code>\box_if_exist_p:N</code> *	<code>\box_if_exist_p:N <box></code>
<code>\box_if_exist_p:c</code> *	<code>\box_if_exist:NTF <box> {(true code)} {(false code)}</code>
<code>\box_if_exist:NTF</code> *	Tests whether the <code><box></code> is currently defined. This does not check that the <code><box></code> 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.

TeXhackers note: This is the TeX 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 TeX. 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.

TeXhackers note: This is the TeX 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

`\box_dp:N` `\box_dp:N` $\langle box \rangle$
`\box_dp:c`
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`.

`\box_ht:N` `\box_ht:N` $\langle box \rangle$
`\box_ht:c`
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`.

`\box_wd:N` `\box_wd:N` $\langle box \rangle$
`\box_wd:c`
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`.

`\box_set_dp:Nn` `\box_set_dp:Nn` $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$
`\box_set_dp:cn`
Updated: 2011-10-22
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.

`\box_set_ht:Nn` `\box_set_ht:Nn` $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$
`\box_set_ht:cn`
Updated: 2011-10-22
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.

`\box_set_wd:Nn` `\box_set_wd:Nn` $\langle box \rangle$ $\{ \langle dimension expression \rangle \}$
`\box_set_wd:cn`
Updated: 2011-10-22
Set the width of the $\langle box \rangle$ to the value of the $\{ \langle dimension expression \rangle \}$. This is a global assignment.

4 Box conditionals

`\box_if_empty_p:N` * `\box_if_empty_p:N` $\langle box \rangle$
`\box_if_empty_p:c` * `\box_if_empty:N`TF $\langle box \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$
`\box_if_empty:N`TF * Tests if $\langle box \rangle$ is a empty (equal to `\c_empty_box`).
`\box_if_empty:c`TF *

`\box_if_horizontal_p:N` * `\box_if_horizontal_p:N` $\langle box \rangle$
`\box_if_horizontal_p:c` * `\box_if_horizontal:N`TF $\langle box \rangle$ $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$
`\box_if_horizontal:N`TF * Tests if $\langle box \rangle$ is a horizontal box.
`\box_if_horizontal:c`TF *

<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.
<code>Updated: 2012-11-04</code>	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>	
<code>Updated: 2012-11-04</code>	

<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.
<code>Updated: 2012-05-11</code>	

<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.
<code>New: 2012-05-11</code>	

<code>\box_log:N</code>	<code>\box_log:N</code> $\langle box \rangle$
<code>\box_log:c</code>	Writes full details of the content of the $\langle box \rangle$ to the log.
<hr/>	
New: 2012-05-11	

<code>\box_log:Nnn</code>	<code>\box_log:Nnn</code> $\langle box \rangle$ $\langle intexpr_1 \rangle$ $\langle intexpr_2 \rangle$
<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.
<hr/>	
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</code> $\langle contents \rangle$
<hr/>	
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</code> $\langle dimexpr \rangle$ $\langle contents \rangle$
<hr/>	
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</code> $\langle contents \rangle$
<hr/>	
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</code> $\langle box \rangle$ $\langle contents \rangle$
<code>\hbox_set:cn</code>	
<code>\hbox_gset:Nn</code>	Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$.
<code>\hbox_gset:cn</code>	
<hr/>	
Updated: 2017-04-05	

<code>\hbox_set_to_wd:Nnn</code>	<code>\hbox_set_to_wd:Nnn</code> $\langle box \rangle$ $\langle dimexpr \rangle$ $\langle contents \rangle$
<code>\hbox_set_to_wd:cnn</code>	
<code>\hbox_gset_to_wd:Nnn</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:cnn</code>	
<hr/>	
Updated: 2017-04-05	

<code>\hbox_overlap_right:n</code>	<code>\hbox_overlap_right:n</code> $\langle contents \rangle$
<hr/>	
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.

<code>\hbox_overlap_left:n</code>	<code>\hbox_overlap_left: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 left of the insertion point.

<code>\hbox_set:Nw</code>	<code>\hbox_set:Nw ⟨box⟩ ⟨contents⟩ \hbox_set_end:</code>
<code>\hbox_set:cw</code>	
<code>\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.
<code>\hbox_gset:Nw</code>	
<code>\hbox_gset:cw</code>	
<code>\hbox_gset_end:</code>	
Updated: 2017-04-05	

<code>\hbox_set_to_wd:Nnw</code>	<code>\hbox_set_to_wd:Nnw ⟨box⟩ {⟨dimexpr⟩} ⟨contents⟩ \hbox_set_end:</code>
<code>\hbox_set_to_wd:cnw</code>	
<code>\hbox_gset_to_wd:Nnw</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
<code>\hbox_gset_to_wd:cnw</code>	
New: 2017-06-08	

<code>\hbox_unpack:N</code>	<code>\hbox_unpack:N ⟨box⟩</code>
<code>\hbox_unpack:c</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 `\unhcopy`.

<code>\hbox_unpack_clear:N</code>	<code>\hbox_unpack_clear:N ⟨box⟩</code>
<code>\hbox_unpack_clear:c</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 `\unhbox`.

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.

<code>\vbox:n</code>	<code>\vbox:n {⟨contents⟩}</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.

<code>\vbox_top:n</code>	<code>\vbox_top:n {⟨contents⟩}</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.

`\vbox_to_ht:n` `\vbox_to_ht:nn <dimexpr> <contents>`
Updated: 2017-04-05 Typesets the `<contents>` into a vertical box of height `<dimexpr>` and then includes this box in the current list for typesetting.

`\vbox_to_zero:n` `\vbox_to_zero:n <contents>`
Updated: 2017-04-05 Typesets the `<contents>` into a vertical box of zero height and then includes this box in the current list for typesetting.

`\vbox_set:Nn` `\vbox_set:Nn <box> <contents>`
`\vbox_set:cn` Typesets the `<contents>` at natural height and then stores the result inside the `<box>`.
`\vbox_gset:Nn`
`\vbox_gset:cn`
Updated: 2017-04-05

`\vbox_set_top:Nn` `\vbox_set_top:Nn <box> <contents>`
`\vbox_set_top:cn` Typesets the `<contents>` at natural height and then stores the result inside the `<box>`. The
`\vbox_gset_top:Nn` baseline of the box is equal to that of the *first* item added to the box.
`\vbox_gset_top:cn`
Updated: 2017-04-05

`\vbox_set_to_ht:Nnn` `\vbox_set_to_ht:Nnn <box> <dimexpr> <contents>`
`\vbox_set_to_ht:cn` Typesets the `<contents>` to the height given by the `<dimexpr>` and then stores the result
`\vbox_gset_to_ht:Nnn` inside the `<box>`.
`\vbox_gset_to_ht:cn`
Updated: 2017-04-05

`\vbox_set:Nw` `\vbox_set:Nw <box> <contents> \vbox_set_end:`
`\vbox_set:cw` Typesets the `<contents>` at natural height and then stores the result inside the `<box>`. In
`\vbox_set_end:` contrast to `\vbox_set:Nn` this function does not absorb the argument when finding the
`\vbox_gset:Nw` `<content>`, and so can be used in circumstances where the `<content>` may not be a simple
`\vbox_gset:cw` argument.
`\vbox_gset_end:`
Updated: 2017-04-05

`\vbox_set_to_ht:Nnw` `\vbox_set_to_wd:Nnw <box> <dimexpr> <contents> \vbox_set_end:`
`\vbox_set_to_ht:cnw` Typesets the `<contents>` to the height given by the `<dimexpr>` and then stores the result
`\vbox_gset_to_ht:Nnw` inside the `<box>`. In contrast to `\vbox_set_to_ht:Nnn` this function does not absorb the
`\vbox_gset_to_ht:cnw` argument when finding the `<content>`, and so can be used in circumstances where the
New: 2017-06-08 `<content>` may not be a simple argument

`\vbox_set_split_to_ht:NNn` `\vbox_set_split_to_ht:NNn <box1> <box2> <dimexpr>`
Updated: 2011-10-22 Sets `<box1>` to contain material to the height given by the `<dimexpr>` by removing content
from the top of `<box2>` (which must be a vertical box).

TeXhackers note: This is the TeX primitive `\vsplit`.

`\vbox_unpack:N` `\vbox_unpack:N` $\langle box \rangle$
`\vbox_unpack:c` Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set.

T_EXhackers note: This is the T_EX primitive `\unvcopy`.

`\vbox_unpack_clear:N` `\vbox_unpack:N` $\langle box \rangle$
`\vbox_unpack_clear:c` Unpacks the content of the vertical $\langle box \rangle$, retaining any stretching or shrinking applied when the $\langle box \rangle$ was set. The $\langle box \rangle$ 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.

`\box_autosize_to_wd_and_ht:Nnn` `\box_autosize_to_wd_and_ht:Nnn` $\langle box \rangle$ $\langle x-size \rangle$ $\langle y-size \rangle$
`\box_autosize_to_wd_and_ht:Nnn`

New: 2017-04-04

Resizes the $\langle box \rangle$ to fit within the given $\langle x-size \rangle$ (horizontally) and $\langle y-size \rangle$ (vertically); both of the sizes are dimension expressions. The $\langle y-size \rangle$ is the height only: 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. The final size of the $\langle box \rangle$ is the smaller of $\langle x-size \rangle$ and $\langle y-size \rangle$, *i.e.* the result fits within the dimensions specified. 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 T_EX group level.

`\box_autosize_to_wd_and_ht_plus_dp:Nnn` `\box_autosize_to_wd_and_ht_plus_dp:Nnn` $\langle box \rangle$ $\langle x-size \rangle$
`\box_autosize_to_wd_and_ht_plus_dp:Nnn` $\langle y-size \rangle$

New: 2017-04-04

Resizes the $\langle box \rangle$ to fit within the given $\langle x-size \rangle$ (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. The final size of the $\langle box \rangle$ is the smaller of $\langle x-size \rangle$ and $\langle y-size \rangle$, *i.e.* the result fits within the dimensions specified. 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 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 $\text{T}_{\text{E}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ 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 $\text{T}_{\text{E}}\text{X}$ group level.

```
\box_resize_to_wd_and_ht_plus_dp:Nnn \box_resize_to_wd_and_ht_plus_dp:Nnn <box> {(x-size)} {(y-size)}
\box_resize_to_wd_and_ht_plus_dp:cnn
```

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.

```
\box_rotate:Nn \box_rotate:Nn <box> {(angle)}
\box_rotate:cnn
```

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.

```
\box_scale:Nnn \box_scale:Nnn <box> {(x-scale)} {(y-scale)}
\box_scale:cnn
```

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

```
\if_hbox:N * \if_hbox:N <box>
  <true code>
\else:
  <false code>
\fi:
```

Tests is $\langle box \rangle$ is a horizontal box.

\TeX hackers note: This is the \TeX primitive `\ifhbox`.

```
\if_vbox:N * \if_vbox:N <box>
  <true code>
\else:
  <false code>
\fi:
```

Tests is $\langle box \rangle$ 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

`\coffin_new:N`
`\coffin_new:c`
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.

`\coffin_clear:N`
`\coffin_clear:c`
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.

`\coffin_set_eq:NN`
`\coffin_set_eq:(Nc|cN|cc)`
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.

`\coffin_if_exist_p:N` *
`\coffin_if_exist_p:c` *
`\coffin_if_exist:NTF` *
`\coffin_if_exist:cTF` *
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.

`\hcoffin_set:Nn`
`\hcoffin_set:cn`
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.

`\hcoffin_set:Nw`
`\hcoffin_set:cw`
`\hcoffin_set_end:`
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.

<code>\vcoffin_set:Nnn</code> <code>\vcoffin_set:cnn</code> <small>New: 2011-08-17</small> <small>Updated: 2012-05-22</small>	<code>\vcoffin_set:Nnn <coffin> {<width>} {<material>}</code> 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.
--	---

<code>\vcoffin_set:Nnw</code> <code>\vcoffin_set:cnw</code> <code>\vcoffin_set:end:</code> <small>New: 2011-09-10</small> <small>Updated: 2012-05-22</small>	<code>\vcoffin_set:Nnw <coffin> {<width>} <material> \vcoffin_set_end:</code> 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.
--	---

<code>\coffin_set_horizontal_pole:Nnn</code> <code>\coffin_set_horizontal_pole:cnn</code> <small>New: 2012-07-20</small>	<code>\coffin_set_horizontal_pole:Nnn <coffin></code> <code>{<pole>} {<offset>}</code>
--	---

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.

<code>\coffin_set_vertical_pole:Nnn</code> <code>\coffin_set_vertical_pole:cnn</code> <small>New: 2012-07-20</small>	<code>\coffin_set_vertical_pole:Nnn <coffin> {<pole>} {<offset>}</code>
--	---

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

<code>\coffin_attach:NnnNnnnn</code> <code>\coffin_attach:(cnnNnnnn Nnncnnnn cnnccnnnn)</code>	<code>\coffin_attach:NnnNnnnn</code> <code><coffin₁₁-pole₁₁-pole_{2 <code><coffin₂₂-pole₁₂-pole_{2 <code>{<x-offset>} {<y-offset>}</code>}</code>}</code>
---	---

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.

<code>\coffin_join:NnnNnnnn</code> <code>\coffin_join:(cnnNnnnn Nnncnnnn cnnccnnnn)</code>	<code>\coffin_join:NnnNnnnn</code> <code>\coffin_join:\langle coffin_1 \rangle \{\langle coffin_1-pole_1 \rangle\} \{\langle coffin_1-pole_2 \rangle\}</code> <code>\coffin_join:\langle coffin_2 \rangle \{\langle coffin_2-pole_1 \rangle\} \{\langle coffin_2-pole_2 \rangle\}</code> <code>\coffin_join:\langle x-offset \rangle \{\langle y-offset \rangle\}</code>
---	---

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.

<code>\coffin_typeset:Nnnnn</code> <code>\coffin_typeset:cnnnn</code>	<code>\coffin_typeset:Nnnnn \langle coffin \rangle \{\langle pole_1 \rangle\} \{\langle pole_2 \rangle\}</code> <code>\coffin_typeset:cnnnn \{\langle x-offset \rangle\} \{\langle y-offset \rangle\}</code>
--	---

Updated: 2012-07-20

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

<code>\coffin_dp:N</code> <code>\coffin_dp:c</code>	<code>\coffin_dp:N \langle coffin \rangle</code>
--	--

Calculates the depth (below the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

<code>\coffin_ht:N</code> <code>\coffin_ht:c</code>	<code>\coffin_ht:N \langle coffin \rangle</code>
--	--

Calculates the height (above the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

<code>\coffin_wd:N</code> <code>\coffin_wd:c</code>	<code>\coffin_wd:N \langle coffin \rangle</code>
--	--

Calculates the width of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

5 Coffin diagnostics

<code>\coffin_display_handles:Nn</code> <code>\coffin_display_handles:cn</code>	<code>\coffin_display_handles:Nn \langle coffin \rangle \{\langle color \rangle\}</code>
--	--

Updated: 2011-09-02

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.

`\coffin_mark_handle:Nnnn`
`\coffin_mark_handle:cnnn`
Updated: 2011-09-02

`\coffin_mark_handle:Nnnn` $\langle coffin \rangle$ $\{pole_1\}$ $\{pole_2\}$ $\{color\}$

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.

`\coffin_show_structure:N`
`\coffin_show_structure:c`
Updated: 2015-08-01

`\coffin_show_structure:N` $\langle coffin \rangle$

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.

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.

`\coffin_log_structure:N`
`\coffin_log_structure:c`
New: 2014-08-22
Updated: 2015-08-01

`\coffin_log_structure:N` $\langle coffin \rangle$

This function writes the structural information about the $\langle coffin \rangle$ in the log file. See also `\coffin_show_structure:N` which displays the result in the terminal.

5.1 Constants and variables

`\c_empty_coffin`

A permanently empty coffin.

`\l_tmpa_coffin`
`\l_tmpb_coffin`

New: 2012-06-19

Scratch coffins 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.

Part XXVIII

The `l3color` 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`.

1.1 Internal functions

`\l__color_current_tl`

New: 2017-06-15
Updated: 2017-10-02

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` $\langle gray \rangle$ Grayscale color with the $\langle gray \rangle$ value running from 0 (fully black) to 1 (fully white)
- `cmymk` $\langle cyan \rangle$ $\langle magenta \rangle$ $\langle yellow \rangle$ $\langle black \rangle$
- `rgb` $\langle red \rangle$ $\langle green \rangle$ $\langle blue \rangle$

Notice that the value are separated by spaces. There is a fourth pre-defined model using a string value and a numerical one:

- `spot` $\langle name \rangle$ $\langle tint \rangle$ A pre-defined spot color, where the $\langle name \rangle$ should be a pre-defined string color name and the $\langle tint \rangle$ 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.

TeXhackers 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.

Part XXIX

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)ptex 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

`\sys_if_output_dvi_p: *` `\sys_if_output_dvi:TF` `{\true_code}` `{\false_code}`

`\sys_if_output_dvi:TF *`

`\sys_if_output_pdf_p: *`

`\sys_if_output_pdf:TF *`

New: 2015-09-19

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.

`\c_sys_output_str`

New: 2015-09-19

The current output mode given as a lower case string: one of `dvi` or `pdf`.

Part XXX

The l3deprecation package

Deprecation errors

1 l3deprecation documentation

A few commands have had to be deprecated over the years. This module defines deprecated and deleted commands to produce an error.

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

<code>\debug_on:n</code>	<code>\debug_on:n { <comma-separated list> }</code>
<code>\debug_off:n</code>	<code>\debug_off:n { <comma-separated list> }</code>

New: 2017-07-16
Updated: 2017-08-02

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;
- `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_ΕX 2_ε package mode loaded with `enable-debug` or another option implying it.

<code>\mode_leave_vertical:</code>	<code>\mode_leave_vertical:</code>
------------------------------------	------------------------------------

New: 2017-07-04

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_ΕX 2_ε `\leavevmode` approach, no box is used by the method implemented here.

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 *<box>* in the output so that only material inside the bounding box is displayed in the output. The updated *<box>* is an hbox, irrespective of the nature of the *<box>* before the clipping is applied. The clipping applies within the current T_EX group level.

These functions require the L^AT_ΕX3 native drivers: they do not work with the L^AT_ΕX 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.

`\box_trim:Nnnnn`
`\box_trim:cnnnn`

`\box_trim:Nnnnn` $\langle box \rangle$ $\{\langle left \rangle\}$ $\{\langle bottom \rangle\}$ $\{\langle right \rangle\}$ $\{\langle top \rangle\}$

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 *dimension expressions*. 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 TeX group level. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

`\box_viewport:Nnnnn`
`\box_viewport:cnnnn`

`\box_viewport:Nnnnn` $\langle box \rangle$ $\{\langle llx \rangle\}$ $\{\langle lly \rangle\}$ $\{\langle urx \rangle\}$ $\{\langle ury \rangle\}$

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 *dimension expressions*. 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 TeX group level.

4 Additions to l3clist

`\clist_rand_item:N` ★
`\clist_rand_item:c` ★
`\clist_rand_item:n` ★

`\clist_rand_item:N` $\langle clist var \rangle$
`\clist_rand_item:n` $\{\langle comma list \rangle\}$

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 is only available in pdfTeX and LuaTeX.

New: 2016-12-06

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.

5 Additions to l3coffins

`\coffin_resize:Nnn`
`\coffin_resize:cnn`

`\coffin_resize:Nnn` $\langle coffin \rangle$ $\{\langle width \rangle\}$ $\{\langle total-height \rangle\}$

Resized the $\langle coffin \rangle$ to $\langle width \rangle$ and $\langle total-height \rangle$, both of which should be given as dimension expressions.

`\coffin_rotate:Nn`
`\coffin_rotate:cn`

`\coffin_rotate:Nn` $\langle coffin \rangle$ $\{\langle angle \rangle\}$

Rotates the $\langle coffin \rangle$ by the given $\langle angle \rangle$ (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.

`\coffin_scale:Nnn`
`\coffin_scale:cnn`

`\coffin_scale:Nnn` $\langle coffin \rangle$ $\{\langle x-scale \rangle\}$ $\{\langle y-scale \rangle\}$

Scales the $\langle coffin \rangle$ by a factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

6 Additions to l3file

`\file_get_md5five_hash:nN`

New: 2017-07-11

`\file_get_md5five_hash:nN` $\{ \langle file\ name \rangle \}$ $\langle str\ var \rangle$

Searches for $\langle file\ name \rangle$ using the current T_EX search path and the additional paths controlled by `\file_path_include:n`. If found, sets the $\langle str\ var \rangle$ 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_EX 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 $\langle str\ var \rangle$ 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 T_EX search path and the additional paths controlled by `\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.

T_EXhackers note: The X_ƒT_EX 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 T_EX search path and the additional paths controlled by `\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.

T_EXhackers note: The X_ƒT_EX 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 T_EX search path and the additional paths controlled by `\file_path_include:n`. If found then reads in the file as additional L^AT_EX 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, `\msg_critical:nn(nn)` should be preferred.

T_EXhackers note: This function must be used on a line on its own: T_EX 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!

7 Additions to l3int

`\int_rand:nn` ★ `\int_rand:nn` $\{\langle integer\ expression_1 \rangle\}$ $\{\langle integer\ expression_2 \rangle\}$

New: 2016-12-06

Evaluates the two $\langle integer\ expressions \rangle$ and produces a pseudo-random number between the two (with bounds included). This is only available in pdfTeX and LuaTeX.

8 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.

`\msg_expandable_error:nnnnnn` ★ `\msg_expandable_error:nnnnnn` $\{\langle module \rangle\}$ $\{\langle message \rangle\}$ $\{\langle arg\ one \rangle\}$ $\{\langle arg\ two \rangle\}$ $\{\langle arg\ three \rangle\}$ $\{\langle arg\ four \rangle\}$
`\msg_expandable_error:nnffff` ★
`\msg_expandable_error:nnnnn` ★
`\msg_expandable_error:nnffff` ★
`\msg_expandable_error:nnnn` ★
`\msg_expandable_error:nnff` ★
`\msg_expandable_error:nnn` ★
`\msg_expandable_error:nnf` ★
`\msg_expandable_error:nn` ★

New: 2015-08-06

Issues an “Undefined error” message from TeX itself using the undefined control sequence `\::error` then prints “! $\langle module \rangle$: ” $\langle error\ message \rangle$, 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.

9 Additions to l3prop

`\prop_count:N` ★ `\prop_count:N` $\langle property\ list \rangle$

`\prop_count:c` ★

Leaves the number of key–value pairs in the $\langle property\ list \rangle$ in the input stream as an $\langle integer\ denotation \rangle$.

`\prop_map_tokens:Nn` ☆
`\prop_map_tokens:cn` ☆

`\prop_map_tokens:Nn` $\langle property\ list \rangle$ $\{ \langle code \rangle \}$

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.

`\prop_rand_key_value:N` ☆
`\prop_rand_key_value:c` ☆

`\prop_rand_key_value:N` $\langle prop\ var \rangle$

Selects a pseudo-random key–value pair in the $\langle property\ list \rangle$ and returns $\{ \langle key \rangle \} \{ \langle value \rangle \}$. If the $\langle property\ list \rangle$ is empty the result is empty. This is only available in pdf \TeX and Lua \TeX .

New: 2016-12-06

\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.

10 Additions to l3seq

`\seq_mapthread_function:NNN` ☆
`\seq_mapthread_function:(NcN|cNN|ccN)` ☆

`\seq_mapthread_function:NNN` $\langle seq_1 \rangle$ $\langle seq_2 \rangle$ $\langle function \rangle$

Applies $\langle function \rangle$ to every pair of items $\langle seq_1\ item \rangle$ – $\langle seq_2\ item \rangle$ from the two sequences, returning items from both sequences from left to right. The $\langle function \rangle$ 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).

`\seq_set_filter:NNn`
`\seq_gset_filter:NNn`

`\seq_set_filter:NNn` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{ \langle inline\ boolexpr \rangle \}$

Evaluates the $\langle inline\ boolexpr \rangle$ for every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline\ boolexpr \rangle$ receives the $\langle item \rangle$ as `#1`. The sequence of all $\langle items \rangle$ for which the $\langle inline\ boolexpr \rangle$ evaluated to `true` is assigned to $\langle sequence_1 \rangle$.

\TeX hackers note: Contrarily to other mapping functions, `\seq_map_break`: cannot be used in this function, and would lead to low-level \TeX errors.

`\seq_set_map:NNn`
`\seq_gset_map:NNn`

`\seq_set_map:NNn` $\langle sequence_1 \rangle$ $\langle sequence_2 \rangle$ $\{ \langle inline\ function \rangle \}$

Applies $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence_2 \rangle$. The $\langle inline\ function \rangle$ should consist of code which will receive the $\langle item \rangle$ as `#1`. The sequence resulting from `x`-expanding $\langle inline\ function \rangle$ applied to each $\langle item \rangle$ is assigned to $\langle sequence_1 \rangle$. As such, the code in $\langle inline\ function \rangle$ should be expandable.

New: 2011-12-22

\TeX hackers note: Contrarily to other mapping functions, `\seq_map_break`: cannot be used in this function, and would lead to low-level \TeX errors.

`\seq_rand_item:N` ★
`\seq_rand_item:c` ★
New: 2016-12-06

`\seq_rand_item:N` $\langle seq\ var \rangle$

Selects a pseudo-random item of the $\langle sequence \rangle$. If the $\langle sequence \rangle$ is empty the result is empty. This is only available in pdfTeX and LuaTeX.

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 Additions to l3skip

`\skip_split_finite_else_action:nnNN` $\langle skipexpr \rangle$ $\langle action \rangle$
 $\langle dimen_1 \rangle$ $\langle dimen_2 \rangle$

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.

12 Additions to l3sys

`\sys_if_rand_exist:p` ★
`\sys_if_rand_exist:TF` ★
New: 2017-05-27

`\sys_if_rand_exist:p`
`\sys_if_rand_exist:TF` $\langle true\ code \rangle$ $\langle false\ code \rangle$

Tests if the engine has a pseudo-random number generator. Currently this is the case in pdfTeX and LuaTeX.

`\sys_rand_seed` ★
New: 2017-05-27

`\sys_rand_seed`:

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.

`\sys_gset_rand_seed:n`
New: 2017-05-27

`\sys_gset_rand_seed:n` $\langle intexpr \rangle$

Sets the seed for the engine's pseudo-random number generator to the $\langle integer\ expression \rangle$. The assignment is global. This random seed affects all `\..._rand` functions (such as `\int_rand:nn` or `\clist_rand_item:n`) as well as other packages relying on the engine's random number generator. Currently only the absolute value of the seed is used. In engines without random number support this produces an error.

`\c_sys_shell_escape_int`
New: 2017-05-27

This variable exposes the internal triple of the shell escape status. The possible values are

- 0 Shell escape is disabled
- 1 Unrestricted shell escape is enabled
- 2 Restricted shell escape is enabled

```
\sys_if_shell_p: * \sys_if_shell_p:  
\sys_if_shell:TF * \sys_if_shell:TF {\true code} {\false code}
```

New: 2017-05-27

Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.

```
\sys_if_shell_unrestricted_p: * \sys_if_shell_unrestricted_p:  
\sys_if_shell_unrestricted:TF * \sys_if_shell_unrestricted:TF {\true code} {\false code}
```

New: 2017-05-27

Performs a check for whether *unrestricted* shell escape is enabled.

```
\sys_if_shell_restricted_p: * \sys_if_shell_restricted_p:  
\sys_if_shell_restricted:TF * \sys_if_shell_restricted:TF {\true code} {\false code}
```

New: 2017-05-27

Performs a check for whether *restricted* 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 `\sys_if_shell:`.

```
\sys_shell_now:n \sys_shell_now:n {\tokens}  
\sys_shell_now:x
```

New: 2017-05-27

Execute `<tokens>` through shell escape immediately.

```
\sys_shell_shipout:n \sys_shell_shipout:n {\tokens}  
\sys_shell_shipout:x
```

New: 2017-05-27

Execute `<tokens>` through shell escape at shipout.

13 Additions to l3tl

```
\tl_if_single_token_p:n * \tl_if_single_token_p:n {\token list}  
\tl_if_single_token:nTF * \tl_if_single_token:nTF {\true code} {\false 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.

```
\tl_reverse_tokens:n * \tl_reverse_tokens:n {\tokens}
```

This function, which works directly on \TeX 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.

\TeX hackers 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.

`\tl_count_tokens:n` * `\tl_count_tokens:n {<tokens>}`

Counts the number of TeX 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>*.

`\tl_lower_case:n` * `\tl_upper_case:n` {<tokens>
`\tl_upper_case:n` * `\tl_upper_case:nn` {<language>} {<tokens>
`\tl_mixed_case:n` *
`\tl_lower_case:nn` *
`\tl_upper_case:nn` *
`\tl_mixed_case:nn` *

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 $\text{\LaTeX} 2_{\epsilon}$ the commands `\cite`, `\ensuremath`, `\label` and `\ref` are automatically included in the list for exclusion from case changing.

`\l_t1_case_change_accents_t1`

This list specifies accent commands which should be left unexpanded in the output. This allows for example

```
\t1_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, `\t1_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.

```
\t1_mixed_case:n { hello~WORLD } % => "Hello world"  
\t1_mixed_case:n { ~hello~WORLD } % => " Hello world"  
\t1_mixed_case:n { {hello}~WORLD } % => "{Hello} world"
```

When finding the first “letter” for this process, any content in math mode or covered by `\l_t1_case_change_exclude_t1` 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.)

`\l_t1_mixed_case_ignore_t1`

The list of characters to ignore when searching for the first “letter” in mixed-casing is determined by `\l_t1_mixed_change_ignore_t1`. 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 `XƎTeX` 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 `Ãd` 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 pdf \TeX .

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> {<setup>} {<filename>}
\tl_set_from_file:cnn
\tl_gset_from_file:Nnn
\tl_gset_from_file:cnn
```

Defines $\langle tl \rangle$ to the contents of $\langle filename \rangle$. Category codes may need to be set appropriately via the $\langle setup \rangle$ argument.

New: 2014-06-25

```
\tl_set_from_file_x:Nnn <tl> {<setup>} {<filename>}
\tl_set_from_file_x:cnn
\tl_gset_from_file_x:Nnn
\tl_gset_from_file_x:cnn
```

Defines $\langle tl \rangle$ to the contents of $\langle filename \rangle$, expanding the contents of the file as it is read. Category codes and other definitions may need to be set appropriately via the $\langle setup \rangle$ argument.

New: 2014-06-25

```
\tl_rand_item:N * \tl_rand_item:N <tl var>
\tl_rand_item:c * \tl_rand_item:n {<token list>}
\tl_rand_item:n *
```

Selects a pseudo-random item of the $\langle token list \rangle$. If the $\langle token list \rangle$ is blank, the result is empty. This is only available in pdf \TeX and Lua \TeX .

New: 2016-12-06

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

`\tl_range:nnn` ★

New: 2017-02-17

Updated: 2017-07-15

`\tl_range:Nnn` $\langle \text{tl var} \rangle$ $\{\langle \text{start index} \rangle\}$ $\{\langle \text{end index} \rangle\}$
`\tl_range:nnn` $\{\langle \text{token list} \rangle\}$ $\{\langle \text{start index} \rangle\}$ $\{\langle \text{end index} \rangle\}$

Leaves in the input stream the items from the $\langle \text{start index} \rangle$ to the $\langle \text{end index} \rangle$ inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Positive $\langle \text{indices} \rangle$ are counted from the start of the $\langle \text{token list} \rangle$, 1 being the first item, and negative $\langle \text{indices} \rangle$ are counted from the end of the token list, -1 being the last item. If either of $\langle \text{start index} \rangle$ or $\langle \text{end index} \rangle$ 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 $\langle \text{start index} \rangle$ must always be smaller than or equal to the $\langle \text{end index} \rangle$: 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`.

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

```

\TlRangeBraced:Nnn * \TlRangeBraced:Nnn <tl var> {<start index>} {<end index>}
\TlRangeBraced:cnn * \TlRangeBraced:nnn {<token list>} {<start index>} {<end index>}
\TlRangeBraced:nnn * \TlRangeUnbraced:Nnn <tl var> {<start index>} {<end index>}
\TlRangeUnbraced:Nnn * \TlRangeUnbraced:nnn {<token list>} {<start index>} {<end index>}
\TlRangeUnbraced:cnn *
\TlRangeUnbraced:nnn *

```

New: 2017-07-15

Leaves in the input stream the items from the $\langle start\ index\rangle$ to the $\langle end\ index\rangle$ inclusive, using the same indexing as `\TlRange:nnn`. Spaces are ignored. Regardless of whether items appear with or without braces in the $\langle token\ list\rangle$, the `\TlRangeBraced:nnn` function wraps each item in braces, while `\TlRangeUnbraced:nnn` does not (overall it removes an outer set of braces). For instance,

```

\iow_term:x { \TlRangeBraced:nnn { abcd~{e}}f } { 2 } { 5 } }
\iow_term:x { \TlRangeBraced:nnn { abcd~{e}}f } { -4 } { -1 } }
\iow_term:x { \TlRangeBraced:nnn { abcd~{e}}f } { -2 } { -1 } }
\iow_term:x { \TlRangeBraced: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

```

\iow_term:x { \TlRangeUnbraced:nnn { abcd~{e}}f } { 2 } { 5 } }
\iow_term:x { \TlRangeUnbraced:nnn { abcd~{e}}f } { -4 } { -1 } }
\iow_term:x { \TlRangeUnbraced:nnn { abcd~{e}}f } { -2 } { -1 } }
\iow_term:x { \TlRangeUnbraced:nnn { abcd~{e}}f } { 0 } { -1 } }

```

prints `bcde`, `cde`, `e`, and an empty line to the terminal. Because braces are removed, the result of `\TlRangeUnbraced:nnn` may have a different number of items as for `\TlRange:nnn` or `\TlRangeBraced:nnn`. In cases where preserving spaces is important, consider the slower function `\TlRange:nnn`.

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.

14 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).

```

\peek_N_type:TF

```

Updated: 2012-12-20

```

\peek_N_type:TF {<true code>} {<false code>}

```

Tests if the next $\langle token\rangle$ in the input stream can be safely grabbed as an N-type argument. The test is $\langle false\rangle$ if the next $\langle token\rangle$ 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 $\langle true\rangle$ in all other cases. Note that a $\langle true\rangle$ result ensures that the next $\langle token\rangle$ is a valid N-type argument. However, if the next $\langle token\rangle$ is for instance `\c_space_token`, the test takes the $\langle false\rangle$ branch, even though the next $\langle token\rangle$ is in fact a valid N-type argument. 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).

Part XXXII

The `l3luatex` package

LuaTeX-specific functions

1 Breaking out to Lua

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 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.

1.1 TeX code interfaces

`\lua_now_x:n` ★ `\lua_now:n` `{⟨token list⟩}`

`\lua_now:n` ★

New: 2015-06-29

The `⟨token list⟩` 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 `⟨Lua input⟩` 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 `⟨Lua input⟩` 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.

`\lua_shipout_x:n` `\lua_shipout:n` `{⟨token list⟩}`

`\lua_shipout:n`

New: 2015-06-30

The `⟨token list⟩` 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 `⟨Lua input⟩` 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 `⟨Lua input⟩` during the page-building routine: no TeX expansion of the `⟨Lua input⟩` 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 `⟨Lua input⟩` is stored as a “whatsit”.

`\lua_escape_x:n` ★ `\lua_escape:n` {*token list*}

`\lua_escape:n` ★

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 \TeX in an x-type manner *but* the function remains fully expandable.

\TeX hackers note: `\lua_escape_x:n` is a macro wrapper around `\luaescapestring`: when $\text{Lua}\TeX$ is in use two expansions are required to yield the result of the Lua code.

1.2 Lua interfaces

As well as interfaces for \TeX , there are a small number of Lua functions provided here. Currently these are intended for internal use only.

`l3kernel.charcat` `l3kernel.charcat(<charcode>, <catcode>)`

Constructs a character of *charcode* and *catcode* and returns the result to \TeX .

`l3kernel.filemdfivesum` `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 \TeX behaviour. If the *file* is not found, nothing is returned with *no error raised*.

`l3kernel.filemoddate` `l3kernel.filemoddate(<file>)`

Returns the of the date/time of last modification of the *file* in the format `D:<year><month><day><hour><minutes>` 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*.

`l3kernel.filesize` `l3kernel.filesize(<file>)`

Returns the size of the *file* in bytes. If the *file* is not found, nothing is returned with *no error raised*.

`l3kernel.strcmp` `l3kernel.strcmp(<str one>, <str two>)`

Compares the two strings and returns 0 to \TeX if the two are identical.

Part XXXIII

The l3drivers package

Drivers

TeX 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* pdfTeX and LuaTeX (no separate driver is used in this case: the engine deals with PDF creation itself).
- `dvips`: The `dvips` program, which works in conjugation with pdfTeX or LuaTeX in DVI mode.
- `dvipdfmx`: The `dvipdfmx` program, which works in conjugation with pdfTeX or LuaTeX in DVI mode.
- `dvisvgm`: The `dvisvgm` program, which works in conjugation with pdfTeX or LuaTeX when run in DVI mode as well as with (u)pTeX and X_YTeX.
- `xdvipdfmx`: The driver used by X_YTeX.

The code here is all very low-level, and should not in general be used outside of the kernel. It is also important to note that many of the functions here are closely tied to the immediate level “up”, and they must be used in the correct contexts.

1 Box clipping

`_driver_box_use_clip:N`

New: 2011-11-11

`_driver_box_use_clip:N <box>`

Inserts the content of the `<box>` at the current insertion point such that any material outside of the bounding box is not displayed by the driver. The material in the `<box>` is still placed in the output stream: the clipping takes place at a driver level.

This function should only be used within a surrounding horizontal box construct.

2 Box rotation and scaling

`_driver_box_use_rotate:Nn` `_driver_box_use_rotate:Nn <box> {<angle>}`

New: 2016-05-12

Inserts the content of the `<box>` at the current insertion point rotated by the `<angle>` (expressed in degrees). The material is inserted with no apparent height or width, and is rotated such the the TeX 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 box to be correct at the TeX side.

This function should only be used within a surrounding horizontal box construct.

`_driver_box_use_scale:Nnn` `_driver_box_use_scale:Nnn` $\langle box \rangle$ $\langle x-scale \rangle$ $\langle y-scale \rangle$

New: 2016-05-12

Inserts the content of the $\langle box \rangle$ at the current insertion point scale by the $\langle x-scale \rangle$ and $\langle y-scale \rangle$. The material is inserted with no apparent height or width. It is the responsibility of the code using this function to adjust the apparent size of the box to be correct at the \TeX side.

This function should only be used within a surrounding horizontal box construct.

3 Color support

`_driver_color_select:n` `_driver_color_select:n` $\langle color \rangle$

`_driver_color_select:V`

New: 2017-10-25

Selects the $\langle color \rangle$ (which is given in low-level format: a $\langle model \rangle$ followed by a space and one or more space-separated axes).

`_driver_color_pickup:N` `_driver_color_pickup:N` $\langle tl \rangle$

New: 2017-10-25

In $\text{\LaTeX} 2_{\epsilon}$ package mode, collects data on the current color from `\current@color` and stores it in the low-level format used by `expl3` in the $\langle tl \rangle$.

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.

`_driver_draw_begin:` `_driver_draw_begin:`

`_driver_draw_end:` $\langle content \rangle$

`_driver_draw_end:`

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 $\langle content \rangle$ should be zero from the \TeX 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).

`_driver_draw_scope_begin:` `_driver_draw_scope_begin:`

`_driver_draw_scope_end:` $\langle content \rangle$

`_driver_draw_scope_end:`

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 \TeX groups and may not be aligned with them.

4.1 Path construction

`_driver_draw_moveto:nn` `_driver_draw_move:nn` $\langle x \rangle$ $\langle y \rangle$

Moves the current drawing reference point to $(\langle x \rangle, \langle y \rangle)$; any active transformation matrix applies.

`_driver_draw_lineto:nn` `_driver_draw_lineto:nn` $\langle x \rangle$ $\langle y \rangle$

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.

`_driver_draw_curveto:nmnmnn` `_driver_draw_curveto:nmnmnn` $\langle x_1 \rangle$ $\langle y_1 \rangle$
 $\langle x_2 \rangle$ $\langle y_2 \rangle$ $\langle x_3 \rangle$ $\langle y_3 \rangle$

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.

`_driver_draw_rectangle:nmnn` `_driver_draw_rectangle:nmnn` $\langle x \rangle$ $\langle y \rangle$ $\langle width \rangle$ $\langle height \rangle$

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.

`_driver_draw_closepath:` `_driver_draw_closepath:`

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

`_driver_draw_stroke:` $\langle path\ construction \rangle$
`_driver_draw_closestroke:` `_driver_draw_stroke:`

Draws a line along the current path, which is also closed by `_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.

`_driver_draw_fill:`
`_driver_draw_fillstroke:`

<path construction>
`_driver_draw_fill:`

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.

`_driver_draw_nonzero_rule:` `_driver_draw_nonzero_rule:`
`_driver_draw_evenodd_rule:`

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.

`_driver_draw_clip:` *<path construction>*
`_driver_draw_clip:`

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.

`_driver_draw_discardpath:` *<path construction>*
`_driver_draw_discardpath:`

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

`_driver_draw_linewidth:n`

`_driver_draw_linewidth:n` *<dimexpr>*

Sets the width to be used for stroking to *<dimexpr>*.

`_driver_draw_dash:nn`

`_driver_draw_dash:nn` *<dash pattern>* *<phase>*

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.*

`_driver_draw_cap_but`: `_driver_draw_cap_but`:
`_driver_draw_cap_rect`:
`_driver_draw_cap_r`:

Sets the style of terminal stroke position to one of butt, rectangle or round.

`_driver_draw_join_b`: `_driver_draw_cap_but`:
`_driver_draw_join_m`: Sets the style of stroke joins to one of bevel, miter or round.
`_driver_draw_join_r`:

`_driver_draw_miterlimit:n` `_driver_draw_miterlimit:n` `{<dimexpr>}`

Sets the miter limit of lines joined as a miter, as described in the PDF and PostScript manuals.

4.4 Color

`_driver_draw_color_cmyk:n` `_driver_draw_color_cmyk:n` `{<cyan>}` `{<magenta>}` `{<yellow>}`
`_driver_draw_color_cmyk_fill:n` `{<black>}`
`_driver_draw_color_cmyk_stroke:n`

Sets the color for drawing to the CMYK values specified, all of which are fp expressions which should evaluate to between 0 and 1. The `fill` and `stroke` versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

`_driver_draw_color_gray:n` `_driver_draw_color_gray:n` `{<gray>}`
`_driver_draw_color_gray_fill:n`
`_driver_draw_color_gray_stroke:n`

Sets the color for drawing to the grayscale value specified, which is fp expressions which should evaluate to between 0 and 1. The `fill` and `stroke` versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

`_driver_draw_color_rgb:n` `_driver_draw_color_rgb:n` `{<red>}` `{<green>}` `{<blue>}`
`_driver_draw_color_rgb_fill:n`
`_driver_draw_color_rgb_stroke:n`

Sets the color for drawing to the RGB values specified, all of which are fp expressions which should evaluate to between 0 and 1. The `fill` and `stroke` versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

4.5 Inserting T_EX material

```
\\_driver_draw_hbox:Nnnnnn \\_driver_draw_hbox:Nnnnnn <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_transformcm:nnnnn \\_driver_draw_transformcm:nnnnn {<a>} {<b>} {<c>} {<d>}  
{<x>} {<y>}
```

Applies the transformation matrix $[abcd]$ and offset vector (x, y) to the current graphic state. This affects any subsequent items in the same scope but not those already given.