

The L^AT_EX3 Interfaces

The L^AT_EX3 Project*

July 28, 2013

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>T_EX</code> concepts not supported by <code>L^AT_EX3</code>	5
II	The <code>l3bootstrap</code> package: Bootstrap code	6
1	Using the <code>L^AT_EX3</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>L^AT_EX3</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	13
3.4	Copying control sequences	15
3.5	Deleting control sequences	16
3.6	Showing control sequences	16
3.7	Converting to and from control sequences	17
4	Using or removing tokens and arguments	18
4.1	Selecting tokens from delimited arguments	20

5	Predicates and conditionals	20
5.1	Tests on control sequences	21
5.2	Testing string equality	22
5.3	Engine-specific conditionals	23
5.4	Primitive conditionals	23
6	Internal kernel functions	24
V	The <code>l3expan</code> package: Argument expansion	27
1	Defining new variants	27
2	Methods for defining variants	28
3	Introducing the variants	28
4	Manipulating the first argument	29
5	Manipulating two arguments	30
6	Manipulating three arguments	31
7	Unbraced expansion	32
8	Preventing expansion	32
9	Internal functions and variables	34
VI	The <code>l3prg</code> package: Control structures	35
1	Defining a set of conditional functions	35
2	The boolean data type	37
3	Boolean expressions	39
4	Logical loops	40
5	Producing n copies	41
6	Detecting <code>TeX</code> 's mode	41
7	Primitive conditionals	42
8	Internal programming functions	42

VII	The <code>l3quark</code> package: Quarks	44
1	Introduction to quarks and scan marks	44
	1.1 Quarks	44
2	Defining quarks	45
3	Quark tests	45
4	Recursion	46
5	Clearing quarks away	47
6	An example of recursion with quarks	47
7	Internal quark functions	48
8	Scan marks	48
VIII	The <code>l3token</code> package: Token manipulation	49
1	All possible tokens	49
2	Character tokens	50
3	Generic tokens	53
4	Converting tokens	54
5	Token conditionals	54
6	Peeking ahead at the next token	58
7	Decomposing a macro definition	61
IX	The <code>l3int</code> package: Integers	62
1	Integer expressions	62
2	Creating and initialising integers	63
3	Setting and incrementing integers	64
4	Using integers	65
5	Integer expression conditionals	65

6	Integer expression loops	67
7	Integer step functions	69
8	Formatting integers	69
9	Converting from other formats to integers	71
10	Viewing integers	72
11	Constant integers	73
12	Scratch integers	73
13	Primitive conditionals	74
14	Internal functions	74
X	The <code>l3skip</code> package: Dimensions and skips	76
1	Creating and initialising <code>dim</code> variables	76
2	Setting <code>dim</code> variables	77
3	Utilities for dimension calculations	77
4	Dimension expression conditionals	78
5	Dimension expression loops	80
6	Using <code>dim</code> expressions and variables	81
7	Viewing <code>dim</code> variables	82
8	Constant dimensions	82
9	Scratch dimensions	82
10	Creating and initialising <code>skip</code> variables	83
11	Setting <code>skip</code> variables	83
12	Skip expression conditionals	84
13	Using <code>skip</code> expressions and variables	84
14	Viewing <code>skip</code> variables	85

15	Constant skips	85
16	Scratch skips	85
17	Inserting skips into the output	86
18	Creating and initialising muskip variables	86
19	Setting muskip variables	87
20	Using muskip expressions and variables	87
21	Viewing muskip variables	88
22	Constant muskips	88
23	Scratch muskips	88
24	Primitive conditional	88
25	Internal functions	89
XI	The l3tl package: Token lists	90
1	Creating and initialising token list variables	91
2	Adding data to token list variables	92
3	Modifying token list variables	92
4	Reassigning token list category codes	93
5	Reassigning token list character codes	93
6	Token list conditionals	94
7	Mapping to token lists	96
8	Using token lists	97
9	Working with the content of token lists	98
10	The first token from a token list	99
11	Viewing token lists	102
12	Constant token lists	103

13	Scratch token lists	103
14	Internal functions	103
XII	The l3seq package: Sequences and stacks	104
1	Creating and initialising sequences	104
2	Appending data to sequences	105
3	Recovering items from sequences	105
4	Recovering values from sequences with branching	106
5	Modifying sequences	107
6	Sequence conditionals	108
7	Mapping to sequences	108
8	Using the content of sequences directly	110
9	Sequences as stacks	110
10	Constant and scratch sequences	112
11	Viewing sequences	112
12	Internal sequence functions	112
XIII	The l3clist package: Comma separated lists	113
1	Creating and initialising comma lists	113
2	Adding data to comma lists	114
3	Modifying comma lists	115
4	Comma list conditionals	115
5	Mapping to comma lists	116
6	Using the content of comma lists directly	119
7	Comma lists as stacks	119
8	Viewing comma lists	121

9	Constant and scratch comma lists	121
XIV	The l3prop package: Property lists	122
1	Creating and initialising property lists	122
2	Adding entries to property lists	123
3	Recovering values from property lists	123
4	Modifying property lists	124
5	Property list conditionals	124
6	Recovering values from property lists with branching	124
7	Mapping to property lists	125
8	Viewing property lists	126
9	Scratch property lists	127
10	Constants	127
11	Internal property list functions	127
XV	The l3box package: Boxes	128
1	Creating and initialising boxes	128
2	Using boxes	129
3	Measuring and setting box dimensions	129
4	Box conditionals	130
5	The last box inserted	131
6	Constant boxes	131
7	Scratch boxes	131
8	Viewing box contents	131
9	Horizontal mode boxes	132
10	Vertical mode boxes	133

11	Primitive box conditionals	135
XVI	The l3coffins package: Coffin code layer	136
1	Creating and initialising coffins	136
2	Setting coffin content and poles	136
3	Joining and using coffins	138
4	Measuring coffins	138
5	Coffin diagnostics	139
	5.1 Constants and variables	139
XVII	The l3color package: Colour support	140
1	Colour in boxes	140
XVIII	The l3msg package: Messages	141
1	Creating new messages	141
2	Contextual information for messages	142
3	Issuing messages	143
4	Redirecting messages	145
5	Low-level message functions	146
6	Kernel-specific functions	147
7	Expandable errors	148
8	Internal l3msg functions	149
XIX	The l3keys package: Key–value interfaces	150
1	Creating keys	151
2	Sub-dividing keys	155
3	Choice and multiple choice keys	155

4	Setting keys	157
5	Handling of unknown keys	158
6	Selective key setting	159
7	Utility functions for keys	160
8	Low-level interface for parsing key-val lists	160
XX	The l3file package: File and I/O operations	162
1	File operation functions	162
1.1	Input-output stream management	163
1.2	Reading from files	164
2	Writing to files	165
2.1	Wrapping lines in output	166
2.2	Constant input-output streams	167
2.3	Primitive conditionals	167
2.4	Internal file functions and variables	167
2.5	Internal input-output functions	168
XXI	The l3fp package: floating points	169
1	Creating and initialising floating point variables	170
2	Setting floating point variables	170
3	Using floating point numbers	171
4	Floating point conditionals	172
5	Floating point expression loops	173
6	Some useful constants, and scratch variables	174
7	Floating point exceptions	175
8	Viewing floating points	176
9	Floating point expressions	177
9.1	Input of floating point numbers	177
9.2	Precedence of operators	178
9.3	Operations	178

10	Disclaimer and roadmap	183
XXII	The l3luatex package: LuaTeX-specific functions	186
1	Breaking out to Lua	186
2	Category code tables	187
XXIII	The l3candidates package: Experimental additions to l3kernel	189
1	Additions to l3basics	189
2	Additions to l3box	189
	2.1 Affine transformations	189
	2.2 Viewing part of a box	190
	2.3 Internal variables	191
3	Additions to l3clist	192
4	Additions to l3coffins	192
5	Additions to l3file	193
6	Additions to l3fp	194
7	Additions to l3prop	194
8	Additions to l3seq	195
9	Additions to l3skip	196
10	Additions to l3tl	197
11	Additions to l3tokens	198
	Index	200

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 *csname*, and indicates that the argument will be turned into a *csname* before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`.
- V and v** These mean *value of variable*. The **V** and **v** specifiers are used to get the content of a variable without needing to worry about the underlying `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 *csname* is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.
- o** This means *expansion once*. In general, the **V** and **v** specifiers are favoured over **o** for recovering stored information. However, **o** is useful for correctly processing information with delimited arguments.

- x** The **x** specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The `\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.

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.

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;

int Integer-valued count register.

prop Property list.

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

skip “Rubber” lengths.

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

tl Token list variables: placeholder for a token list.

1.1 Terminological inexactitude

A word of warning. In this document, and others referring to the **expl3** programming modules, we often refer to “variables” and “functions” as if they were actual constructs from a real programming language. In truth, **T_EX** 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 **T_EX**’s stomach” (if you are familiar with the **T_EX**book parlance).

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

2 Documentation conventions

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

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

`\ExplSyntaxOn`
`\ExplSyntaxOff`

`\ExplSyntaxOn ... \ExplSyntaxOff`

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

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

`\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 it to be used within an `x`-type argument (in plain T_EX 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:

<code>\xetex_if_engine:<i>TF</i> *</code>	<code>\xetex_if_engine:TF {\langle true code \rangle} {\langle false code \rangle}</code>
---	---

The underlining and italic of `TF` indicates that `\xetex_if_engine:T`, `\xetex_if_engine:F` and `\xetex_if_engine: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:

<code>\l_tmpa_tl</code>	A short piece of text will describe the variable: there is no syntax illustration in this case. In some cases, the function is similar to one in $\text{\LaTeX} 2_\epsilon$ or plain \TeX . In these cases, the text will include an extra “ \TeXhackers note ” section:
-------------------------	---

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

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`.

3 Formal language conventions which apply generally

As this is a formal reference guide for $\text{\LaTeX} 3$ programming, the descriptions of functions are intended to be reasonably “complete”. However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

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

4 \TeX concepts not supported by $\text{\LaTeX} 3$

The \TeX concept of an “`\outer`” macro is *not supported* at all by $\text{\LaTeX} 3$. As such, the functions provided here may break when used on top of $\text{\LaTeX} 2_\epsilon$ if `\outer` tokens are used in the arguments.

Part II

The l3bootstrap package

Bootstrap code

1 Using the L^AT_EX3 modules

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

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

`\ExplSyntaxOn`
`\ExplSyntaxOff`

Updated: 2011-08-13

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

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

`\ProvidesExplPackage`
`\ProvidesExplClass`
`\ProvidesExplFile`

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

These functions act broadly in the same way as the 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.)

`\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 alike 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

`_expl_package_check:`

`_expl_package_check:`

Used to ensure that all parts of `expl3` are loaded together (*i.e.* as part of `expl3`). Issues an error if a kernel package is loaded independently of the bundle.

`\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 L^AT_EX3 programming language

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

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

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

`\tex_...` Introduced by T_EX itself;

`\etex_...` Introduced by the ε -T_EX extensions;

`\pdftex_...` Introduced by pdfT_EX;

`\xetex_...` Introduced by X_YT_EX;

`\luatex_...` Introduced by LuaT_EX.

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_begin:`**

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

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

`\group_insert_after:N`**`\group_insert_after:N`** *(token)*

Adds *(token)* to the list of *(tokens)* to be inserted when the current group level ends. The list of *(tokens)* to be inserted will be empty at the beginning of a group: multiple applications of `\group_insert_after:N` may be used to build the inserted list one *(token)* at a time. The current group level may be closed by a `\group_end:` function or by a token with category code 2 (close-group). The later will be 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” will be fully expanded inside an **x** expansion. In contrast, “protected” functions are not expanded within **x** expansions.

3.1 Defining functions

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

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

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

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

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

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

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

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

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

N and n No manipulation.

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

p and **w** These are special cases.

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

3.2 Defining new functions using parameter text

```
\cs_new:Npn
\cs_new:(cpn|Npx|cpx)
```

```
\cs_new:Npn <function> <parameters> {<code>}
```

Creates $\langle function \rangle$ 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 definition is global and an error will result if the $\langle function \rangle$ is already defined.

```
\cs_new_nopar:Npn
\cs_new_nopar:(cpn|Npx|cpx)
```

```
\cs_new_nopar:Npn <function> <parameters> {<code>}
```

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

```
\cs_new_protected:Npn
\cs_new_protected:(cpn|Npx|cpx)
```

```
\cs_new_protected:Npn <function> <parameters> {<code>}
```

Creates $\langle function \rangle$ 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 $\langle function \rangle$ will not expand within an x-type argument. The definition is global and an error will result if the $\langle function \rangle$ is already defined.

```
\cs_new_protected_nopar:Npn
\cs_new_protected_nopar:(cpn|Npx|cpx)
```

```
\cs_new_protected_nopar:Npn <function> <parameters> {<code>}
```

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

```
\cs_set:Npn
\cs_set:(cpn|Npx|cpx)
```

```
\cs_set:Npn <function> <parameters> {<code>}
```

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

`\cs_set_nopar:Npn`
`\cs_set_nopar:(cpn|Npx|cpx)`

`\cs_set_nopar:Npn <function> <parameters> {<code>}`

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

`\cs_set_protected:Npn`
`\cs_set_protected:(cpn|Npx|cpx)`

`\cs_set_protected:Npn <function> <parameters> {<code>}`

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

`\cs_set_protected_nopar:Npn`
`\cs_set_protected_nopar:(cpn|Npx|cpx)`

`\cs_set_protected_nopar:Npn <function> <parameters> {<code>}`

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

`\cs_gset:Npn`
`\cs_gset:(cpn|Npx|cpx)`

`\cs_gset:Npn <function> <parameters> {<code>}`

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

`\cs_gset_nopar:Npn`
`\cs_gset_nopar:(cpn|Npx|cpx)`

`\cs_gset_nopar:Npn <function> <parameters> {<code>}`

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

`\cs_gset_protected:Npn`
`\cs_gset_protected:(cpn|Npx|cpx)`

`\cs_gset_protected:Npn <function> <parameters> {<code>}`

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

$\backslash\text{cs_gset_protected_nopar:Npn}$ $\backslash\text{cs_gset_protected_nopar: (cpn Npx cpx)}$	$\backslash\text{cs_gset_protected_nopar:Npn } \langle\text{function}\rangle \langle\text{parameters}\rangle \{ \langle\text{code}\rangle \}$
---	--

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

3.3 Defining new functions using the signature

$\backslash\text{cs_new:Nn}$ $\backslash\text{cs_new: (cn Nx cx)}$	$\backslash\text{cs_new:Nn } \langle\text{function}\rangle \{ \langle\text{code}\rangle \}$
---	--

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

$\backslash\text{cs_new_nopar:Nn}$ $\backslash\text{cs_new_nopar: (cn Nx cx)}$	$\backslash\text{cs_new_nopar:Nn } \langle\text{function}\rangle \{ \langle\text{code}\rangle \}$
---	---

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

$\backslash\text{cs_new_protected:Nn}$ $\backslash\text{cs_new_protected: (cn Nx cx)}$	$\backslash\text{cs_new_protected:Nn } \langle\text{function}\rangle \{ \langle\text{code}\rangle \}$
---	---

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

$\backslash\text{cs_new_protected_nopar:Nn}$ $\backslash\text{cs_new_protected_nopar: (cn Nx cx)}$	$\backslash\text{cs_new_protected_nopar:Nn } \langle\text{function}\rangle \{ \langle\text{code}\rangle \}$
---	--

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

<hr/> <code>\cs_set:Nn</code> <hr/> <code>\cs_set:(cn Nx cx)</code>	<code>\cs_set:Nn <function> {<code>}</code> Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (<i>#1, #2, etc.</i>) will be replaced by those absorbed by the function. The assignment of a meaning to the <i><function></i> is restricted to the current \TeX group level.
<hr/> <code>\cs_set_nopar:Nn</code> <hr/> <code>\cs_set_nopar:(cn Nx cx)</code>	<code>\cs_set_nopar:Nn <function> {<code>}</code> Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (<i>#1, #2, etc.</i>) will be replaced by those absorbed by the function. When the <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.
<hr/> <code>\cs_set_protected:Nn</code> <hr/> <code>\cs_set_protected:(cn Nx cx)</code>	<code>\cs_set_protected:Nn <function> {<code>}</code> Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (<i>#1, #2, etc.</i>) will be replaced by those absorbed by the function. The <i><function></i> will not expand within an x-type argument. The assignment of a meaning to the <i><function></i> is restricted to the current \TeX group level.
<hr/> <code>\cs_set_protected_nopar:Nn</code> <hr/> <code>\cs_set_protected_nopar:(cn Nx cx)</code>	<code>\cs_set_protected_nopar:Nn <function> {<code>}</code> Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (<i>#1, #2, etc.</i>) will be replaced by those absorbed by the function. When the <i><function></i> is used the <i><parameters></i> absorbed cannot contain <code>\par</code> tokens. The <i><function></i> will not expand within an x-type argument. The assignment of a meaning to the <i><function></i> is restricted to the current \TeX group level.
<hr/> <code>\cs_gset:Nn</code> <hr/> <code>\cs_gset:(cn Nx cx)</code>	<code>\cs_gset:Nn <function> {<code>}</code> Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (<i>#1, #2, etc.</i>) will be replaced by those absorbed by the function. The assignment of a meaning to the <i><function></i> is global.
<hr/> <code>\cs_gset_nopar:Nn</code> <hr/> <code>\cs_gset_nopar:(cn Nx cx)</code>	<code>\cs_gset_nopar:Nn <function> {<code>}</code> Sets <i><function></i> to expand to <i><code></i> as replacement text. Within the <i><code></i> , the number of <i><parameters></i> is detected automatically from the function signature. These <i><parameters></i> (<i>#1, #2, etc.</i>) will be replaced by those absorbed by the function. When the <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 global.

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

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

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

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

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

Updated: 2012-01-14

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

3.4 Copying control sequences

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

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

<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.

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

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

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

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

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

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

3.5 Deleting control sequences

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

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

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

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

Updated: 2011-09-15

3.6 Showing control sequences

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

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

This function expands to the *meaning* of the $\langle control\ sequence \rangle$ control sequence. This will show the $\langle replacement\ text \rangle$ for a macro.

\TeX hackers note: This is \TeX 's `\meaning` primitive. The `c` variant correctly reports undefined arguments.

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

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

Displays the definition of the $\langle control\ sequence \rangle$ on the terminal.

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

Updated: 2012-09-09

3.7 Converting to and from control sequences

\use:c ★ \use:c {*<control sequence name>*}

Converts the given *<control sequence name>* into a single control sequence token. This process requires two expansions. The content for *<control sequence name>* may be literal material or from other expandable functions. The *<control sequence name>* must, when fully expanded, consist of character tokens which are not active: typically, they will be 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
\l_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:NTF \cs_if_exist_use:N *<control sequence>*

\cs_if_exist_use:cTF

New: 2012-11-10

Tests whether the *<control sequence>* is currently defined (whether as a function or another control sequence type), and if it does inserts the *<control sequence>* into the input stream.

\cs_if_exist_use:NTF ★

\cs_if_exist_use:cTF ★

New: 2012-11-10

\cs_if_exist_use:N *<control sequence>* {*<true code>*} {*<false code>*}

Tests whether the *<control sequence>* is currently defined (whether as a function or another control sequence type), and if it does inserts the *<control sequence>* into the input stream followed by the *<true code>*.

\cs:w ★

\cs_end: ★

\cs:w *<control sequence name>* \cs_end:

Converts the given *<control sequence name>* into a single control sequence token. This process requires one expansion. The content for *<control sequence name>* may be literal material or from other expandable functions. The *<control sequence name>* must, when fully expanded, consist of character tokens which are not active: typically, they will be 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 sequence will *not* include the current escape token, cf. `\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 will be required to convert the *<control sequence>* to a sequence of characters in the input stream. In most cases, an *f*-expansion will be 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 in absorbing them the outer set will be 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 the situation in force when first function absorbs the token).

```

\use:n ★ \use:n {\group_1}
\use:(nn|nnn|nnnn) ★ \use:nn {\group_1} {\group_2}
\use:nnn {\group_1} {\group_2} {\group_3}
\use:nnnn {\group_1} {\group_2} {\group_3} {\group_4}

```

As illustrated, these functions will absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument will be removed leaving the remaining tokens in the input stream. The category code of these tokens will also be 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 } }
```

will result in the input stream containing

```
abc { def }
```

i.e. only the outer braces will be removed.

<code>\use_i:nn</code>	★	<code>\use_i:nn {\langle arg_1 \rangle} {\langle arg_2 \rangle}</code>
------------------------	---	--

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

<code>\use_i:nnn</code>	★	<code>\use_i:nnn {\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle}</code>
-------------------------	---	---

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

<code>\use_i:nnnn</code>	★	<code>\use_i:nnnn {\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle} {\langle arg_4 \rangle}</code>
--------------------------	---	--

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

<code>\use_i_ii:nnn</code>	★	<code>\use_i_ii:nnn {\langle arg_1 \rangle} {\langle arg_2 \rangle} {\langle arg_3 \rangle}</code>
----------------------------	---	--

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

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

will result in the input stream containing

`abc { def }`

i.e. the outer braces will be removed and the third group will be removed.

<code>\use_none:n</code>	★	<code>\use_none:n {\langle group_1 \rangle}</code>
<code>\use_none:(nn nnn nnnn nnnnn nnnnnn nnnnnnn nnnnnnnn nnnnnnnnn)</code>	★	

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

<hr/> <code>\use:x</code> <hr/>	<code>\use:x {⟨expandable tokens⟩}</code>
Updated: 2011-12-31	Fully expands the <i>⟨expandable tokens⟩</i> 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.

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

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

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

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

5 Predicates and conditionals

L^AT_EX3 has three concepts for conditional flow processing:

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

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

a function that will turn 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 carry 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 will usually be 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 will always provide all three versions.

Important to note is that these branching conditionals with $\langle true\ code \rangle$ and/or $\langle false\ code \rangle$ 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 will be 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
} {\langle true code \rangle} {\langle false code \rangle}
```

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

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

`\c_true_bool`
`\c_false_bool`

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

5.1 Tests on control sequences

<code>\cs_if_eq_p:NN</code> ★	<code>\cs_if_eq_p:NN {\langle cs_1 \rangle} {\langle cs_2 \rangle}</code>
<code>\cs_if_eq:NNTF</code> ★	<code>\cs_if_eq:NNTF {\langle cs_1 \rangle} {\langle cs_2 \rangle} {\langle true code \rangle} {\langle false code \rangle}</code>

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

<code>\cs_if_exist_p:N</code> ★	<code>\cs_if_exist_p:N \langle control sequence \rangle</code>
<code>\cs_if_exist_p:c</code> ★	<code>\cs_if_exist:NNTF \langle control sequence \rangle {\langle true code \rangle} {\langle false code \rangle}</code>
<code>\cs_if_exist:NNTF</code> ★	Tests whether the $\langle control\ sequence \rangle$ is currently defined (whether as a function or another control sequence type). Any valid definition of $\langle control\ sequence \rangle$ will evaluate as <code>true</code> .
<code>\cs_if_exist:cTF</code> ★	

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

5.2 Testing string equality

<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, and is **true** if the two lists 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

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. Thus for example

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

is logically true.

<code>\str_case:nnTF</code>	★	<code>\str_case:nnTF</code>	$\{\langle test\ string \rangle\}$
<code>\str_case:onTF</code>	★	{	
		$\{\langle string\ case_1 \rangle\}$	$\{\langle code\ case_1 \rangle\}$
		$\{\langle string\ case_2 \rangle\}$	$\{\langle code\ case_2 \rangle\}$
		...	
		$\{\langle string\ case_n \rangle\}$	$\{\langle code\ case_n \rangle\}$
		}	
		$\{\langle true\ code \rangle\}$	
		$\{\langle false\ code \rangle\}$	

New: 2013-07-24

This function compares the $\langle test\ string \rangle$ in turn with each of the $\langle string\ cases \rangle$. If the two are equal (as described for `\str_if_eq:nnTF` then the associated $\langle code \rangle$ is left in the input stream. If any of the cases are matched, the $\langle true\ code \rangle$ is also inserted into the input stream (after the code for the appropriate case), while if none match then the $\langle false\ code \rangle$ is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

<code>\str_case_x:nnTF</code> ★	<code>\str_case_x:nnn {<test string>}</code>
New: 2013-07-24	<pre> { {<string case₁>} {<code case₁>} {<string case₂>} {<code case₂>} ... {<string case_n>} {<code case_n>} } {<true code>} {<false code>} </pre>

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

5.3 Engine-specific conditionals

<code>\luatex_if_engine_p:</code> ★	<code>\luatex_if_engine:TF {<true code>} {<false code>}</code>
<code>\luatex_if_engine:TF</code> ★	Detects is the document is being compiled using LuaTeX.
Updated: 2011-09-06	
<code>\pdftex_if_engine_p:</code> ★	<code>\pdftex_if_engine:TF {<true code>} {<false code>}</code>
<code>\pdftex_if_engine:TF</code> ★	Detects is the document is being compiled using pdfTeX.
Updated: 2011-09-06	
<code>\xetex_if_engine_p:</code> ★	<code>\xetex_if_engine:TF {<true code>} {<false code>}</code>
<code>\xetex_if_engine:TF</code> ★	Detects is the document is being compiled using XeTeX.
Updated: 2011-09-06	

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

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

<code>\if_meaning:w</code>	★	<code>\if_meaning:w <arg₁₂</code>
----------------------------	---	---

`\if_meaning:w` executes *<true code>* when *<arg_{1 and *<arg_{2 are the same, otherwise it executes *<false code>*. *<arg_{1 and *<arg_{2 could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.}*}*}*}*

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

<code>\if:w</code>	★	<code>\if:w <token₁₂</code>
<code>\if_catcode:w</code>	★	<code>\if_catcode:w <token₁₂</code>
<code>\if_catcode:w</code>	★	These conditionals will 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 <i><true code></i> if currently in horizontal mode, otherwise execute <i><false code></i> . Similar for the other functions.
<code>\if_mode_math:</code>	★	
<code>\if_mode_inner:</code>	★	

6 Internal kernel functions

<code>__chk_if_exist_cs:N</code>	<code>__chk_if_exist_cs:N <cs></code>
<code>__chk_if_exist_cs:c</code>	This function checks that <i><cs></i> exists according to the criteria for <code>\cs_if_exist_p:N</code> , and if not raises a kernel-level error.

<u>_chk_if_free_cs:N</u> <u>_chk_if_free_cs:c</u>	_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.
<u>_cs_count_signature:N</u> ★ <u>_cs_count_signature:c</u> ★	_cs_count_signature:N $\langle function \rangle$ Splits the $\langle function \rangle$ into the $\langle name \rangle$ (<i>i.e.</i> the part before the colon) and the $\langle signature \rangle$ (<i>i.e.</i> 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.
<u>_cs_split_function:NN</u> ★	_cs_split_function:NN $\langle function \rangle$ $\langle processor \rangle$ Splits the $\langle function \rangle$ into the $\langle name \rangle$ (<i>i.e.</i> the part before the colon) and the $\langle signature \rangle$ (<i>i.e.</i> 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 $\langle name \rangle$ will 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).
<u>_cs_get_function_name:N</u> ★	_cs_get_function_name:N $\langle function \rangle$ Splits the $\langle function \rangle$ into the $\langle name \rangle$ (<i>i.e.</i> the part before the colon) and the $\langle signature \rangle$ (<i>i.e.</i> 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).
<u>_cs_get_function_signature:N</u> ★	_cs_get_function_signature:N $\langle function \rangle$ Splits the $\langle function \rangle$ into the $\langle name \rangle$ (<i>i.e.</i> the part before the colon) and the $\langle signature \rangle$ (<i>i.e.</i> after the colon). The $\langle signature \rangle$ is then left in the input stream made up of tokens with category code 12 (other).
<u>_cs_tmp:w</u>	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).
<u>_kernel_register_show:N</u> <u>_kernel_register_show:c</u>	_kernel_register_show:N $\langle register \rangle$ Used to show the contents of a T _E X register at the terminal, formatted such that internal parts of the mechanism are not visible.
<u>_prg_case_end:nw</u>	_prg_case_end:nw $\{\langle code \rangle\}$ $\langle tokens \rangle$ \q_mark $\{\langle true code \rangle\}$ \q_mark $\{\langle false code \rangle\}$ \q_stop Used to terminate case statements (\int_case:nnTF, <i>etc.</i>) by removing trailing $\langle tokens \rangle$ and the end marker \q_stop, inserting the $\langle code \rangle$ for the successful case (if one is found) and either the true code or false code for the over all outcome, as appropriate.

`_str_if_eq_x_return:nn` `_str_if_eq_x_return:nn {\langle t1 \rangle} {\langle t2 \rangle}`

Compares the full expansion of two *token lists* 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:nn(TF)` coded for speed.

Part V

The l3expan package

Argument expansion

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

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

1 Defining new variants

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

Internally preprocessing of arguments is done with functions from the `\exp_` module. They all look alike, an example would be `\exp_args:NNo`. This function has three arguments, the first and the second are a single tokens, while the third argument should be given in braces. Applying `\exp_args:NNo` will expand 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_new_nopar:Npn \seq_gpush:No { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is uncritical as the `\cs_new_nopar:Npn` function will silently accept definitions whenever the new definition is identical to an already given one. 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

`\cs_generate_variant:Nn`

Updated: 2013-07-09

`\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 will expand its arguments as detailed and pass 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 }
```

will create a new function `\foo:cn` which will expand its first argument into a control sequence name and pass the result to `\foo:Nn`. Similarly

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

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

3 Introducing the variants

The available internal functions for argument expansion come in two flavours, some of them are faster than others. Therefore 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 will not 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` will need special processing which 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`, `num`, `int`, `skip`, `dim`, `toks`, or built-in T_EX registers. The `v` type is the same except it first creates a

control sequence out of its argument before returning the value. This recent addition to the argument specifiers may shake things up a bit as most places where `o` is used will be replaced by `V`. The documentation you are currently reading will therefore require a fair bit of re-writing.

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. Let's pretend we want to set the control sequence whose name is given by `b \l_tmpa_tl b` equal to the list of tokens `\aaa a`. Furthermore we want to store the execution of it in a `<tl var>`. In this example we assume `\l_tmpa_tl` contains the text string `lur`. The straightforward approach is

```
\tl_set:No \l_tmpb_tl { \tl_set:cn { b \l_tmpa_tl b } { \aaa a } }
```

Unfortunately this only puts `\exp_args:Nc \tl_set:Nn {b \l_tmpa_tl b} { \aaa a }` into `\l_tmpb_tl` and not `\tl_set:Nn \blurb { \aaa a }` as we probably wanted. Using `\tl_set:Nx` is not an option as that will die horribly. Instead we can do a

```
\tl_set:Nf \l_tmpb_tl { \tl_set:cn { b \l_tmpa_tl b } { \aaa a } }
```

which puts the desired result in `\l_tmpb_tl`. It requires `\tl_set:Nf` to be defined as

```
\cs_set_nopar:Npn \tl_set:Nf { \exp_args:NNf \tl_set:Nn }
```

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!

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 will be 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 will occur 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 will be left unchanged.

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

<hr/> <hr/>	<hr/> <hr/>
<code>\exp_args:NV</code> ★	<code>\exp_args:NV</code> $\langle function \rangle$ $\langle variable \rangle$
	This function absorbs two arguments (the names of the $\langle function \rangle$ and the $\langle variable \rangle$). The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.
<hr/> <hr/>	<hr/> <hr/>
<code>\exp_args:Nv</code> ★	<code>\exp_args:Nv</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are expanded until only characters remain, and are then turned into a control sequence. (An internal error will occur if such a conversion is not possible). This control sequence should be the name of a $\langle variable \rangle$. The content of the $\langle variable \rangle$ are recovered and placed inside braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.
<hr/> <hr/>	<hr/> <hr/>
<code>\exp_args:Nf</code> ★	<code>\exp_args:Nf</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$). The $\langle tokens \rangle$ are fully expanded until the first non-expandable token or space is found, and the result is inserted in braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.
<hr/> <hr/>	<hr/> <hr/>
<code>\exp_args:Nx</code>	<code>\exp_args:Nx</code> $\langle function \rangle$ $\{\langle tokens \rangle\}$
	This function absorbs two arguments (the $\langle function \rangle$ name and the $\langle tokens \rangle$) and exhaustively expands the $\langle tokens \rangle$ second. The result is inserted in braces into the input stream <i>after</i> reinsertion of the $\langle function \rangle$. Thus the $\langle function \rangle$ may take more than one argument: all others will be left unchanged.

5 Manipulating two arguments

<hr/> <hr/>	<hr/> <hr/>
<code>\exp_args:NNo</code> ★	<code>\exp_args:NNo</code> $\langle token_1 \rangle$ $\langle token_2 \rangle$ $\{\langle tokens \rangle\}$
<code>\exp_args:(NNc NNv NNf Nco Ncf Ncc NVV)</code> ★	
	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.
<hr/> <hr/>	<hr/> <hr/>
<code>\exp_args:Nno</code> ★	<code>\exp_args:Nno</code> $\langle token \rangle$ $\{\langle tokens_1 \rangle\}$ $\{\langle tokens_2 \rangle\}$
<code>\exp_args:(NnV Nnf Noo Nof Noc Nff Nfo Nnc)</code> ★	
	Updated: 2012-01-14

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.

<code>\exp_args:NNx</code>	<code>\exp_args:NNx <token₁> <token₂> {\tokens}</code>
<code>\exp_args:(Nnx Ncx Nox Nxo Nxx)</code>	

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions are not expandable.

6 Manipulating three arguments

<code>\exp_args:NNNo</code>	★	<code>\exp_args:NNNo <token₁> <token₂> <token₃> {\tokens}</code>
<code>\exp_args:(NNNV Nccc NcNc NcNo Ncco)</code>	★	

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

<code>\exp_args:NNoo</code>	★	<code>\exp_args:NNNo <token₁> <token₂> <token₃> {\tokens}</code>
<code>\exp_args:(NNno Nnno Nnnc Nooo)</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, *etc.* These functions need special (slower) processing.

<code>\exp_args:NNnx</code>	<code>\exp_args:NNnx <token₁> <token₂> {\tokens₁} {\tokens₂}</code>
<code>\exp_args:(NNox Nnnx Nnox Noox Ncnx Nccx)</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, *etc.*

7 Unbraced expansion

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

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.

T_EXhackers note: As an optimization, the last argument is unbraced by some of those functions before expansion. This can cause problems if the argument is empty: for instance, `\exp_last_unbraced:Nf \mypkg_foo: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.

<code>\exp_after:wN</code>	★	<code>\exp_after:wN</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$
----------------------------	---	----------------------------	---------------------------	---------------------------

Carries out a single expansion of $\langle token_2 \rangle$ (which may consume arguments) prior to the expansion of $\langle token_1 \rangle$. If $\langle token_2 \rangle$ is a T_EX primitive, it will be executed rather than expanded, while if $\langle token_2 \rangle$ has not expansion (for example, if it is a character) then it will be 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 T_EX category codes). Unless specifically required, expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_arg:N` function.

T_EXhackers note: This is the T_EX 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 will not appear after the expansion has completed.

<hr/> <hr/>	<code>\exp_not:N</code> ★	<code>\exp_not:N</code> $\langle token \rangle$
		Prevents expansion of the $\langle token \rangle$ in a context where it would otherwise be expanded, for example an x -type argument.
		T_EXhackers note: This is the T _E X <code>\noexpand</code> primitive.
<hr/> <hr/>	<code>\exp_not:c</code> ★	<code>\exp_not:c</code> $\{\langle tokens \rangle\}$
		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.
<hr/> <hr/>	<code>\exp_not:n</code> ★	<code>\exp_not:n</code> $\{\langle tokens \rangle\}$
		Prevents expansion of the $\langle tokens \rangle$ in a context where they would otherwise be expanded, for example an x -type argument.
		T_EXhackers note: This is the ε -T _E X <code>\unexpanded</code> primitive. Hence its argument <i>must</i> be surrounded by braces.
<hr/> <hr/>	<code>\exp_not:V</code> ★	<code>\exp_not:V</code> $\langle variable \rangle$
		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.
<hr/> <hr/>	<code>\exp_not:v</code> ★	<code>\exp_not:v</code> $\{\langle tokens \rangle\}$
		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.
<hr/> <hr/>	<code>\exp_not:o</code> ★	<code>\exp_not:o</code> $\{\langle tokens \rangle\}$
		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.
<hr/> <hr/>	<code>\exp_not:f</code> ★	<code>\exp_not:f</code> $\{\langle tokens \rangle\}$
		Expands $\langle tokens \rangle$ 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.
<hr/> <hr/>	<code>\exp_stop_f:</code> ★	<code>\function:f</code> $\langle tokens \rangle$ <code>\exp_stop_f:</code> $\langle more tokens \rangle$
Updated: 2011-06-03		This function terminates an f -type expansion. Thus if a function <code>\function:f</code> starts an f -type expansion and all of $\langle tokens \rangle$ are expandable <code>\exp_stop:f</code> will terminate the expansion of tokens even if $\langle more tokens \rangle$ are also expandable. The function itself is an implicit space token. Inside an x -type expansion, it will retain its form, but when typeset it produces the underlying space (\sqcup).

9 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.

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

`\::N` Internal forms for the base expansion types. These names do *not* conform to the general $\text{\LaTeX}3$ approach as this makes them more readily visible in the log and so forth.

`\::c`

`\::o`

`\::f`

`\::x`

`\::v`

`\::V`

`\:::`

Part VI

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 typical states returned are *⟨true⟩* and *⟨false⟩* but other states are possible, say an *⟨error⟩* state for erroneous input, *e.g.*, text as input in a function comparing integers.

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. Important to note here is that the arguments are executed after exiting the underlying `\if... \fi:` structure.

1 Defining a set of conditional functions

```
\prg_new_conditional:Npnn
\prg_new_conditional:Nnn
\prg_set_conditional:Npnn
\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 will 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_new_protected_conditional:Nnn {<conditions>} {<code>}
\prg_set_protected_conditional:Npnn \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 will check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the **set** version will 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 will not work properly for `protected` conditionals.
- `\<name>:<arg spec>T` — a function with one more argument than the original `<arg spec>` demands. The `<true branch>` code in this additional argument will be left on the input stream only if the test is `true`.
- `\<name>:<arg spec>F` — a function with one more argument than the original `<arg spec>` demands. The `<false branch>` code in this additional argument will be left on the input stream only if the test is `false`.
- `\<name>:<arg spec>TF` — a function with two more argument than the original `<arg spec>` demands. The `<true branch>` code in the first additional argument will be left on the input stream if the test is `true`, while the `<false branch>` code in the second argument will be left on the input stream if the test is `false`.

The `<code>` of the test may use `<parameters>` as specified by the second argument to `\prg_set_conditional:Npnn`: this should match the `<argument specification>` but this is not enforced. The `Nnn` versions infer the number of arguments from the argument specification given (cf. `\cs_new:Nn`, etc.). Within the `<code>`, the functions `\prg_return_true:` and `\prg_return_false:` are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

```
\prg_set_conditional:Npnn \foo_if_bar:NN #1#2 { p , T , TF }
{
  \if_meaning:w \l_tmpa_tl #1
    \prg_return_true:
  \else:
    \if_meaning:w \l_tmpa_tl #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  \fi:
}
```

This defines the function `\foo_if_bar_p:NN`, `\foo_if_bar:NNTF` and `\foo_if_bar:NNT` but not `\foo_if_bar:NNF` (because `F` is missing from the `<conditions>` list). The return statements take care of resolving the remaining `\else:` and `\fi:` before returning the state. There must be a return statement for each branch; failing to do so will result in erroneous output if that branch is executed.

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

These functions copies a family of conditionals. The **new** version will check for existing definitions (*cf.* `\cs_new:Npn`) whereas the **set** version will not (*cf.* `\cs_set:Npn`). The conditionals copied are depended on the comma-separated list of `<conditions>`, which should be one or more of **p**, **T**, **F** and **TF**.

<code>\prg_return_true: ★</code>	<code>\prg_return_true:</code>
<code>\prg_return_false: ★</code>	<code>\prg_return_false:</code>

These ‘return’ functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npnn`, *etc.*, to indicate when a true or false branch has been 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 will generally mean being constructed from predicate functions, possibly nested).

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

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

<code>\bool_set_false:N</code>	<code>\bool_set_false:N <boolean></code>
<code>\bool_set_false:c</code>	
<code>\bool_gset_false:N</code>	
<code>\bool_gset_false:c</code>	Sets <code><boolean></code> logically false .

<hr/> <code>\bool_set_true:N</code> <code>\bool_set_true:c</code> <code>\bool_gset_true:N</code> <code>\bool_gset_true:c</code> <hr/>	<code>\bool_set_true:N</code> $\langle\textit{boolean}\rangle$ Sets $\langle\textit{boolean}\rangle$ logically true.
<hr/> <code>\bool_set_eq:NN</code> <code>\bool_set_eq:(cN Nc cc)</code> <code>\bool_gset_eq:NN</code> <code>\bool_gset_eq:(cN Nc cc)</code> <hr/>	<code>\bool_set_eq:NN</code> $\langle\textit{boolean}_1\rangle$ $\langle\textit{boolean}_2\rangle$ Sets the content of $\langle\textit{boolean}_1\rangle$ equal to that of $\langle\textit{boolean}_2\rangle$.
<hr/> <code>\bool_set:Nn</code> <code>\bool_set:cn</code> <code>\bool_gset:Nn</code> <code>\bool_gset:cn</code> <hr/> Updated: 2012-07-08 <hr/>	<code>\bool_set:Nn</code> $\langle\textit{boolean}\rangle$ $\{\langle\textit{boolexpr}\rangle\}$ Evaluates the $\langle\textit{boolean expression}\rangle$ as described for <code>\bool_if:n(TF)</code> , and sets the $\langle\textit{boolean}\rangle$ variable to the logical truth of this evaluation.
<hr/> <code>\bool_if_p:N</code> ★ <code>\bool_if_p:c</code> ★ <code>\bool_if:NTF</code> ★ <code>\bool_if:cTF</code> ★ <hr/>	<code>\bool_if_p:N</code> $\langle\textit{boolean}\rangle$ <code>\bool_if:NTF</code> $\langle\textit{boolean}\rangle$ $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$ Tests the current truth of $\langle\textit{boolean}\rangle$, and continues expansion based on this result.
<hr/> <code>\bool_show:N</code> <code>\bool_show:c</code> <hr/> New: 2012-02-09 <hr/>	<code>\bool_show:N</code> $\langle\textit{boolean}\rangle$ Displays the logical truth of the $\langle\textit{boolean}\rangle$ on the terminal.
<hr/> <code>\bool_show:n</code> <hr/> New: 2012-02-09 Updated: 2012-07-08 <hr/>	<code>\bool_show:n</code> $\{\langle\textit{boolean expression}\rangle\}$ Displays the logical truth of the $\langle\textit{boolean expression}\rangle$ on the terminal.
<hr/> <code>\bool_if_exist_p:N</code> ★ <code>\bool_if_exist_p:c</code> ★ <code>\bool_if_exist:NTF</code> ★ <code>\bool_if_exist:cTF</code> ★ <hr/> New: 2012-03-03 <hr/>	<code>\bool_if_exist_p:N</code> $\langle\textit{boolean}\rangle$ <code>\bool_if_exist:NTF</code> $\langle\textit{boolean}\rangle$ $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$ Tests whether the $\langle\textit{boolean}\rangle$ is currently defined. This does not check that the $\langle\textit{boolean}\rangle$ really is a boolean variable.
<hr/> <code>\l_tmpa_bool</code> <code>\l_tmpb_bool</code> <hr/>	A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_bool</code> <code>\g_tmpb_bool</code> <hr/>	A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any L ^A T _E X3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean $\langle true \rangle$ or $\langle false \rangle$ values, it seems only fitting that we also provide a parser for $\langle boolean\ expressions \rangle$.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean $\langle true \rangle$ or $\langle false \rangle$. It supports the logical operations And, Or and Not as the well-known infix operators $\&\&$, $||$ and $!$ with their usual precedences. 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 } ||
  \int_compare_p:n { 1 = \error } % is skipped
) &&
! ( \int_compare_p:n { 2 = 4 } )
```

is a valid boolean expression. Note that minimal evaluation is carried out whenever possible so that whenever a truth value cannot be changed any more, the remaining tests within the current group are skipped.

\backslash bool_if_p:n ★ \backslash bool_if:nTF ★	\backslash bool_if_p:n $\{ \langle boolean\ expression \rangle \}$ \backslash bool_if:nTF $\{ \langle boolean\ expression \rangle \} \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
--	---

Updated: 2012-07-08

Tests the current truth of $\langle boolean\ expression \rangle$, and continues expansion based on this result. The $\langle boolean\ expression \rangle$ should consist of a series of predicates or boolean variables with the logical relationship between these defined using $\&\&$ (“And”), $||$ (“Or”), $!$ (“Not”) and parentheses. Minimal evaluation is used in the processing, so that once a result is defined there is not further expansion of the tests. For example

```
\bool_if_p:n
{
  \int_compare_p:nNn { 1 } = { 1 }
  &&
  (
    \int_compare_p:nNn { 2 } = { 3 } ||
    \int_compare_p:nNn { 4 } = { 4 } ||
    \int_compare_p:nNn { 1 } = { \error } % is skipped
  )
  &&
  ! \int_compare_p:nNn { 2 } = { 4 }
}
```

will be **true** and will not evaluate \backslash int_compare_p:nNn { 1 } = { \error }. The logical Not applies to the next predicate or group.

<hr/>	
<code>\bool_not_p:n</code> ☆	<code>\bool_not_p:n {<boolean expression>}</code>
Updated: 2012-07-08	Function version of <code>!(<boolean expression>)</code> within a boolean expression.
<hr/>	
<code>\bool_xor_p:nn</code> ☆	<code>\bool_xor_p:nn {<boolexpr₁>} {<boolexpr₂>}</code>
Updated: 2012-07-08	Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operator.
<hr/>	

4 Logical loops

Loops using either boolean expressions or stored boolean values.

<hr/>	
<code>\bool_do_until:Nn</code> ☆	<code>\bool_do_until:Nn <boolean> {<code>}</code>
<code>\bool_do_until:cn</code> ☆	Places the <code><code></code> in the input stream for T _E X to process, and then checks the logical value of the <code><boolean></code> . If it is <code>false</code> then the <code><code></code> will be inserted into the input stream again and the process will loop until the <code><boolean></code> is <code>true</code> .
<hr/>	
<code>\bool_do_while:Nn</code> ☆	<code>\bool_do_while:Nn <boolean> {<code>}</code>
<code>\bool_do_while:cn</code> ☆	Places the <code><code></code> in the input stream for T _E X to process, and then checks the logical value of the <code><boolean></code> . If it is <code>true</code> then the <code><code></code> will be inserted into the input stream again and the process will loop until the <code><boolean></code> is <code>false</code> .
<hr/>	
<code>\bool_until_do:Nn</code> ☆	<code>\bool_until_do:Nn <boolean> {<code>}</code>
<code>\bool_until_do:cn</code> ☆	This function firsts checks the logical value of the <code><boolean></code> . If it is <code>false</code> the <code><code></code> is placed in the input stream and expanded. After the completion of the <code><code></code> the truth of the <code><boolean></code> is re-evaluated. The process will then loop until the <code><boolean></code> is <code>true</code> .
<hr/>	
<code>\bool_while_do:Nn</code> ☆	<code>\bool_while_do:Nn <boolean> {<code>}</code>
<code>\bool_while_do:cn</code> ☆	This function firsts checks the logical value of the <code><boolean></code> . If it is <code>true</code> the <code><code></code> is placed in the input stream and expanded. After the completion of the <code><code></code> the truth of the <code><boolean></code> is re-evaluated. The process will then loop until the <code><boolean></code> is <code>false</code> .
<hr/>	
<code>\bool_do_until:nn</code> ☆	<code>\bool_do_until:nn {<boolean expression>} {<code>}</code>
Updated: 2012-07-08	Places the <code><code></code> in the input stream for T _E X to process, and then checks the logical value of the <code><boolean expression></code> as described for <code>\bool_if:nTF</code> . If it is <code>false</code> then the <code><code></code> will be inserted into the input stream again and the process will loop until the <code><boolean expression></code> evaluates to <code>true</code> .
<hr/>	
<code>\bool_do_while:nn</code> ☆	<code>\bool_do_while:nn {<boolean expression>} {<code>}</code>
Updated: 2012-07-08	Places the <code><code></code> in the input stream for T _E X to process, and then checks the logical value of the <code><boolean expression></code> as described for <code>\bool_if:nTF</code> . If it is <code>true</code> then the <code><code></code> will be inserted into the input stream again and the process will loop until the <code><boolean expression></code> evaluates to <code>false</code> .
<hr/>	

<hr/> <code>\bool_until_do:nn</code> ☆	<code>\bool_until_do:nn {\boolean expression} {\code}</code>
Updated: 2012-07-08	This function firsts checks the logical value of the <i>boolean expression</i> (as described for <code>\bool_if:nTF</code>). If it is false the <i>code</i> is placed in the input stream and expanded. After the completion of the <i>code</i> the truth of the <i>boolean expression</i> is re-evaluated. The process will then loop until the <i>boolean expression</i> is true .

<hr/> <code>\bool_while_do:nn</code> ☆	<code>\bool_while_do:nn {\boolean expression} {\code}</code>
Updated: 2012-07-08	This function firsts checks the logical value of the <i>boolean expression</i> (as described for <code>\bool_if:nTF</code>). If it is true the <i>code</i> is placed in the input stream and expanded. After the completion of the <i>code</i> the truth of the <i>boolean expression</i> is re-evaluated. The process will then loop until the <i>boolean expression</i> is false .

5 Producing n copies

<hr/> <code>\prg_replicate:nn</code> ☆	<code>\prg_replicate:nn {\integer expression} {\tokens}</code>
Updated: 2011-07-04	Evaluates the <i>integer expression</i> (which should be zero or positive) and creates the resulting number of copies of the <i>tokens</i> . The function is both expandable and safe for nesting. It yields its result after two expansion steps.

6 Detecting T_EX's mode

<hr/> <code>\mode_if_horizontal_p:</code> ☆	<code>\mode_if_horizontal_p:</code>
<code>\mode_if_horizontal:TF</code> ☆	<code>\mode_if_horizontal:TF {\true code} {\false code}</code>
	Detects if T _E X is currently in horizontal mode.

<hr/> <code>\mode_if_inner_p:</code> ☆	<code>\mode_if_inner_p:</code>
<code>\mode_if_inner:TF</code> ☆	<code>\mode_if_inner:TF {\true code} {\false code}</code>
	Detects if T _E X is currently in inner mode.

<hr/> <code>\mode_if_math_p:</code> ☆	<code>\mode_if_math:TF {\true code} {\false code}</code>
<code>\mode_if_math:TF</code> ☆	
Updated: 2011-09-05	Detects if T _E X is currently in maths mode.

<hr/> <code>\mode_if_vertical_p:</code> ☆	<code>\mode_if_vertical_p:</code>
<code>\mode_if_vertical:TF</code> ☆	<code>\mode_if_vertical:TF {\true code} {\false code}</code>
	Detects if T _E X is currently in vertical mode.

7 Primitive conditionals

`\if_predicate:w` ★ `\if_predicate:w <predicate> <true code> \else: <false code> \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 `<predicate>` but to make the coding clearer this should be done through `\if_bool:N`.)

`\if_bool:N` ★ `\if_bool:N <boolean> <true code> \else: <false code> \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:` ★
`\group_align_safe_end:`

Updated: 2011-08-11

These functions are used to enclose material in a \TeX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the `&` token inside `\halign`. This is necessary to allow grabbing of tokens for testing purposes, as \TeX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` will 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.

`\scan_align_safe_stop:` `\scan_align_safe_stop:`

Updated: 2011-09-06

Stops \TeX 's scanner looking for expandable control sequences at the beginning of an alignment cell. This function is required, for example, to obtain the expected output when testing `\mode_if_math:TF` at the start of a math array cell: placing `\scan_align_safe_stop:` before `\mode_if_math:TF` will give the correct result. This function does not destroy any kerning if used in other locations, but *does* render functions non-expandable.

\TeX hackers note: This is a protected version of `\prg_do_nothing:`, which therefore stops \TeX 's scanner in the circumstances described without producing any affect on the output.

`__prg_variable_get_scope:N` ★ `__prg_variable_get_scope:N <variable>`

Returns the scope (g for global, blank otherwise) for the `<variable>`.

`__prg_variable_get_type:N` ★ `__prg_variable_get_type:N <variable>`

Returns the type of `<variable>` (tl, int, etc.)

<u><code>__prg_break_point:Nn</code></u> ★	<code>__prg_break_point:Nn \<type>_map_break: <tokens></code> <p>Used to mark the end of a recursion or mapping: the functions <code>\<type>_map_break:</code> and <code>\<type>_map_break:n</code> use this to break out of the loop. After the loop ends, the <code><tokens></code> are inserted into the input stream. This occurs even if the break functions are <i>not</i> applied: <code>__prg_break_point:Nn</code> is functionally-equivalent in these cases to <code>\use_ii:nn</code>.</p>
<u><code>__prg_map_break:Nn</code></u> ★	<code>__prg_map_break:Nn \<type>_map_break: {(user code)}</code> <code>...</code> <code>__prg_break_point:Nn \<type>_map_break: {(ending code)}</code> <p>Breaks a recursion in mapping contexts, inserting in the input stream the <code><user code></code> after the <code><ending code></code> for the loop. The function breaks loops, inserting their <code><ending code></code>, until reaching a loop with the same <code><type></code> as its first argument. This <code>\<type>_map_break:</code> argument is simply used as a recognizable marker for the <code><type></code>.</p>
<u><code>\g__prg_map_int</code></u>	<p>This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions <code>__prg_map_1:w</code>, <code>__prg_map_2:w</code>, <i>etc.</i>, labelled by <code>\g__prg_map_int</code> hold functions to be mapped over various list datatypes in inline and variable mappings.</p>
<u><code>__prg_break_point:</code></u> ★	<p>This copy of <code>\prg_do_nothing:</code> is used to mark the end of a fast short-term recursions: the function <code>__prg_break:n</code> uses this to break out of the loop.</p>
<u><code>__prg_break:</code></u> ★ <u><code>__prg_break:n</code></u> ★	<code>__prg_break:n {(tokens)} ... __prg_break_point:</code> <p>Breaks a recursion which has no <code><ending code></code> and which is not a user-breakable mapping (see for instance <code>\prop_get:Nn</code>), and inserts <code><tokens></code> in the input stream.</p>

Part VII

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, with the most command use case as 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

<u><code>\quark_new:N</code></u>	<code>\quark_new:N <quark></code> Creates a new <code><quark></code> which expands only to <code><quark></code> . The <code><quark></code> will be defined globally, and an error message will be raised if the name was already taken.
<u><code>\q_stop</code></u>	Used as a marker for delimited arguments, such as $\cs_set:Npn \tmp:w \#1\#2 \q_stop \{ \#1 \}$
<u><code>\q_mark</code></u>	Used as a marker for delimited arguments when <code>\q_stop</code> is already in use. Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself may need to be tested (in contrast to <code>\q_stop</code> , which is only ever used as a delimiter).
<u><code>\q_no_value</code></u>	A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a “return” value by functions such as <code>\prop_get:NnN</code> if there is no data to return.

3 Quark tests

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

<u><code>\quark_if_nil_p:N</code> ★</u>	<code>\quark_if_nil_p:N <token></code>
<u><code>\quark_if_nil:NTF</code> ★</u>	<code>\quark_if_nil:NTF <token> {<true code>} {<false code>}</code>
	Tests if the <code><token></code> is equal to <code>\q_nil</code> .
<u><code>\quark_if_nil_p:n</code> ★</u>	<code>\quark_if_nil_p:n {<token list>}</code>
<u><code>\quark_if_nil_p:(o V)</code> ★</u>	<code>\quark_if_nil:nTF {<token list>} {<true code>} {<false code>}</code>
<u><code>\quark_if_nil:nTF</code> ★</u>	Tests if the <code><token list></code> contains only <code>\q_nil</code> (distinct from <code><token list></code> being empty or containing <code>\q_nil</code> plus one or more other tokens).
<u><code>\quark_if_nil:(o V)TF</code> ★</u>	
<u><code>\quark_if_no_value_p:N</code> ★</u>	<code>\quark_if_no_value_p:N <token></code>
<u><code>\quark_if_no_value_p:c</code> ★</u>	<code>\quark_if_no_value:NTF <token> {<true code>} {<false code>}</code>
<u><code>\quark_if_no_value:NTF</code> ★</u>	Tests if the <code><token></code> is equal to <code>\q_no_value</code> .
<u><code>\quark_if_no_value:cTF</code> ★</u>	
<u><code>\quark_if_no_value_p:n</code> ★</u>	<code>\quark_if_no_value_p:n {<token list>}</code>
<u><code>\quark_if_no_value:nTF</code> ★</u>	<code>\quark_if_no_value:NTF {<token list>} {<true code>} {<false code>}</code>
	Tests if the <code><token list></code> contains only <code>\q_no_value</code> (distinct from <code><token list></code> being empty or containing <code>\q_no_value</code> 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 6.

<code>\q_recursion_tail</code>	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.
--------------------------------	---

<code>\q_recursion_stop</code>	This quark is added <i>after</i> the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.
--------------------------------	---

<code>\quark_if_recursion_tail_stop:N</code>	<code>\quark_if_recursion_tail_stop:N <token></code>
--	--

Tests if $\langle token \rangle$ contains only the marker `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

<code>\quark_if_recursion_tail_stop:n</code>	<code>\quark_if_recursion_tail_stop:n {<token list>}</code>
<code>\quark_if_recursion_tail_stop:o</code>	

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ contains only `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

<code>\quark_if_recursion_tail_stop_do:Nn</code>	<code>\quark_if_recursion_tail_stop_do:Nn <token> {<insertion>}</code>
--	--

Tests if $\langle token \rangle$ contains only the marker `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. 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.

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

Updated: 2011-09-06

Tests if the $\langle token list \rangle$ contains only `\q_recursion_tail`, and if so terminates the recursion this is part of using `\use_none_delimit_by_q_recursion_stop:w`. 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 Clearing quarks away

<code>\use_none_delimit_by_q_recursion_stop:w</code>	<code>\use_none_delimit_by_q_recursion_stop:w <tokens></code> <code>\q_recursion_stop</code>
--	---

Used to prematurely terminate a recursion using `\q_recursion_stop` as the end marker, removing any remaining `<tokens>` from the input stream.

<code>\use_i_delimit_by_q_recursion_stop:nw</code>	<code>\use_i_delimit_by_q_recursion_stop:nw {<insertion>}</code> <code><tokens> \q_recursion_stop</code>
--	---

Used to prematurely terminate a recursion using `\q_recursion_stop` as the end marker, removing any remaining `<tokens>` from the input stream. The `<insertion>` is then made into the input stream after the end of the recursion.

6 An example of recursion with quarks

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

Here’s the definition of `\my_map_dbl:nn`. First of all, define the function that will do 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.

```

1 \cs_new:Npn \my_map_dbl:nn #1#2
2   {
3     \cs_set:Npn \__my_map_dbl_fn:nn ##1 ##2 {#2}
4     \__my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail \q_recursion_stop
5   }

```

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.

```

6 \cs_new:Nn \__my_map_dbl:nn
7   {
8     \quark_if_recursion_tail_stop:n {#1}
9     \quark_if_recursion_tail_stop:n {#2}
10    \__my_map_dbl_fn:nn {#1} {#2}

```

Finally, recurse:

```

11    \__my_map_dbl:nn
12  }

```

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

7 Internal quark functions

<code>_quark_if_recursion_tail_break:NN</code>	<code>_quark_if_recursion_tail_break:nN {<token list>}</code>
<code>_quark_if_recursion_tail_break:nN</code>	<code>\<type>_map_break:</code>

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:`.

8 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence will never expand in an expansion context and will be (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.

<code>_scan_new:N</code>	<code>_scan_new:N <scan mark></code>
----------------------------	--

Creates a new `<scan mark>` which is set equal to `\scan_stop:`. The `<scan mark>` will be defined globally, and an error message will be raised if the name was already taken by another scan mark.

<code>\s_stop</code>	Used at the end of a set of instructions, as a marker that can be jumped to using <code>_use_none_delimit_by_s_stop:w</code> .
-----------------------	--

<code>_use_none_delimit_by_s_stop:w</code>	<code>_use_none_delimit_by_s_stop:w <tokens> \s_stop</code>
--	--

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

Part VIII

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 will have two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most of the time we will be using the term “token” but most of the time the function we're describing can equally well be used on a control sequence as such one is one token as well.

We shall refer to list of tokens as `tlists` and such lists represented by a single control sequence is a “token list variable” `tl var`. Functions for these two types are found in the `l3tl` module.

1 All possible tokens

Let us start by reviewing every case that a given token can fall into. It is very important to distinguish two aspects of a token: its meaning, and what it looks like.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three for the same internal operation of T_EX, namely the primitive testing the next two characters for equality of their character code. They 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 will take 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
```

2 Character tokens

<code>\char_set_catcode_escape:N</code>	<code>\char_set_catcode_letter:N</code> $\langle character \rangle$
<code>\char_set_catcode_group_begin:N</code>	
<code>\char_set_catcode_group_end:N</code>	
<code>\char_set_catcode_math_toggle:N</code>	
<code>\char_set_catcode_alignment:N</code>	
<code>\char_set_catcode_end_line:N</code>	
<code>\char_set_catcode_parameter:N</code>	
<code>\char_set_catcode_math_superscript:N</code>	
<code>\char_set_catcode_math_subscript:N</code>	
<code>\char_set_catcode_ignore:N</code>	
<code>\char_set_catcode_space:N</code>	
<code>\char_set_catcode_letter:N</code>	
<code>\char_set_catcode_other:N</code>	
<code>\char_set_catcode_active:N</code>	
<code>\char_set_catcode_comment:N</code>	
<code>\char_set_catcode_invalid:N</code>	

Sets the category code of the $\langle character \rangle$ to that indicated in the function name. Depending on the current category code of the $\langle token \rangle$ the escape token may also be needed:

`\char_set_catcode_other:N \%`

The assignment is local.

<code>\char_set_catcode_escape:n</code>	<code>\char_set_catcode_letter:n</code> $\{ \langle integer\ expression \rangle \}$
<code>\char_set_catcode_group_begin:n</code>	
<code>\char_set_catcode_group_end:n</code>	
<code>\char_set_catcode_math_toggle:n</code>	
<code>\char_set_catcode_alignment:n</code>	
<code>\char_set_catcode_end_line:n</code>	
<code>\char_set_catcode_parameter:n</code>	
<code>\char_set_catcode_math_superscript:n</code>	
<code>\char_set_catcode_math_subscript:n</code>	
<code>\char_set_catcode_ignore:n</code>	
<code>\char_set_catcode_space:n</code>	
<code>\char_set_catcode_letter:n</code>	
<code>\char_set_catcode_other:n</code>	
<code>\char_set_catcode_active:n</code>	
<code>\char_set_catcode_comment:n</code>	
<code>\char_set_catcode_invalid:n</code>	

Sets the category code of the $\langle character \rangle$ which has character code as given by the $\langle integer\ expression \rangle$. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

<hr/> <hr/> <code>\char_set_catcode:nn</code>	<code>\char_set_catcode:nn {⟨integer₁⟩} {⟨integer₂⟩}</code>
	These functions set the category code of the <i>⟨character⟩</i> which has character code as given by the <i>⟨integer expression⟩</i> . The first <i>⟨integer expression⟩</i> is the character code and the second is the category code to apply. The setting applies within the current T _E X group. In general, the symbolic functions <code>\char_set_catcode_⟨type⟩</code> should be preferred, but there are cases where these lower-level functions may be useful.
<hr/> <hr/> <code>\char_value_catcode:n</code> ★	<code>\char_value_catcode:n {⟨integer expression⟩}</code>
	Expands to the current category code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
<hr/> <hr/> <code>\char_show_value_catcode:n</code>	<code>\char_show_value_catcode:n {⟨integer expression⟩}</code>
	Displays the current category code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.
<hr/> <hr/> <code>\char_set_lccode:nn</code>	<code>\char_set_lccode:nn {⟨integer₁⟩} {⟨integer₂⟩}</code>
	This function set up the behaviour of <i>⟨character⟩</i> when found inside <code>\tl_to_lowercase:n</code> , such that <i>⟨character₁⟩</i> will be converted into <i>⟨character₂⟩</i> . The two <i>⟨characters⟩</i> may be specified using an <i>⟨integer expression⟩</i> for the character code concerned. This may include the T _E X ‘ <i>⟨character⟩</i> ’ method for converting a single character into its character code:
	<pre> \char_set_lccode:nn { ‘\A } { ‘\a } % Standard behaviour \char_set_lccode:nn { ‘\A } { ‘\A + 32 } \char_set_lccode:nn { 50 } { 60 } </pre>
	The setting applies within the current T _E X group.
<hr/> <hr/> <code>\char_value_lccode:n</code> ★	<code>\char_value_lccode:n {⟨integer expression⟩}</code>
	Expands to the current lower case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> .
<hr/> <hr/> <code>\char_show_value_lccode:n</code>	<code>\char_show_value_lccode:n {⟨integer expression⟩}</code>
	Displays the current lower case code of the <i>⟨character⟩</i> with character code given by the <i>⟨integer expression⟩</i> on the terminal.

<code>\char_set_uccode:nn</code>	<code>\char_set_uccode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
----------------------------------	--

This function set up the behaviour of $\langle character \rangle$ when found inside `\tl_to_uppercase:n`, such that $\langle character_1 \rangle$ will be converted into $\langle character_2 \rangle$. The two $\langle characters \rangle$ may be specified using an $\langle integer expression \rangle$ for the character code concerned. This may include the T_EX ‘ $\langle character \rangle$ ’ 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 {⟨integer expression⟩}</code>
-------------------------------------	--

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 {⟨integer expression⟩}</code>
--	---

Displays the current upper case code of the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.

<code>\char_set_mathcode:nn</code>	<code>\char_set_mathcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
------------------------------------	--

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_EX group.

<code>\char_value_mathcode:n</code> ★	<code>\char_value_mathcode:n {⟨integer expression⟩}</code>
---------------------------------------	--

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 {⟨integer expression⟩}</code>
--	---

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>	<code>\char_set_sfcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}</code>
----------------------------------	--

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_EX group.

<code>\char_value_sfcode:n</code> ★	<code>\char_value_sfcode:n {⟨integer expression⟩}</code>
-------------------------------------	--

Expands to the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$.

<hr/> <hr/>	<hr/>
<code>\char_show_value_sfcode:n</code>	<code>\char_show_value_sfcode:n {\langle integer expression \rangle}</code>
<hr/>	<hr/>
	Displays the current space factor for the $\langle character \rangle$ with character code given by the $\langle integer expression \rangle$ on the terminal.

<hr/>	<hr/>
<code>\l_char_active_seq</code>	Used to track which tokens will require special handling at the document level as they are of category $\langle active \rangle$ (catcode 13). Each entry in the sequence consists of a single active character. Active tokens should be added to the sequence when they are defined for general document use.
<hr/>	<hr/>
New: 2012-01-23	

<hr/>	<hr/>
<code>\l_char_special_seq</code>	Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories $\langle letter \rangle$ (catcode 11) or $\langle other \rangle$ (catcode 12). Each entry in the sequence consists of a single escaped token, for example <code>\</code> for the backslash or <code>\{</code> for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.
<hr/>	<hr/>
New: 2012-01-23	

3 Generic tokens

<hr/>	<hr/>
<code>\token_new:Nn</code>	<code>\token_new:Nn \langle token_1 \rangle {\langle token_2 \rangle}</code>
<hr/>	<hr/>
	Defines $\langle token_1 \rangle$ to globally be a snapshot of $\langle token_2 \rangle$. This will be an implicit representation of $\langle token_2 \rangle$.

<hr/>	<hr/>
<code>\c_group_begin_token</code> <code>\c_group_end_token</code> <code>\c_math_toggle_token</code> <code>\c_alignment_token</code> <code>\c_parameter_token</code> <code>\c_math_superscript_token</code> <code>\c_math_subscript_token</code> <code>\c_space_token</code>	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.
<hr/>	<hr/>

<hr/>	<hr/>
<code>\c_catcode_letter_token</code> <code>\c_catcode_other_token</code>	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.
<hr/>	<hr/>

<hr/>	<hr/>
<code>\c_catcode_active_tl</code>	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.
<hr/>	<hr/>

4 Converting tokens

<code>\token_to_meaning:N</code>	★	<code>\token_to_meaning:N <token></code>
<code>\token_to_meaning:c</code>	★	

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

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

<code>\token_to_str:N</code>	★	<code>\token_to_str:N <token></code>
<code>\token_to_str:c</code>	★	

Converts the given $\langle token \rangle$ into a series of characters with category code 12 (other). The current escape character will be the first character in the sequence, although this will also have category code 12 (the escape character is part of the $\langle token \rangle$). This function requires only a single expansion.

\TeX hackers note: `\token_to_str:N` is the \TeX primitive `\string` renamed.

5 Token conditionals

<code>\token_if_group_begin_p:N</code>	★	<code>\token_if_group_begin_p:N <token></code>
<code>\token_if_group_begin:NTF</code>	★	<code>\token_if_group_begin:NTF <token> {\true code} {\false code}</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.

<code>\token_if_group_end_p:N</code>	★	<code>\token_if_group_end_p:N <token></code>
<code>\token_if_group_end:NTF</code>	★	<code>\token_if_group_end:NTF <token> {\true code} {\false code}</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.

<code>\token_if_math_toggle_p:N</code>	★	<code>\token_if_math_toggle_p:N <token></code>
<code>\token_if_math_toggle:NTF</code>	★	<code>\token_if_math_toggle:NTF <token> {\true code} {\false code}</code>

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

<code>\token_if_alignment_p:N</code>	★	<code>\token_if_alignment_p:N <token></code>
<code>\token_if_alignment:NTF</code>	★	<code>\token_if_alignment:NTF <token> {\true code} {\false code}</code>

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

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

Tests if `<token>` has the category code of a macro parameter token (`#` when normal `TEX` category codes are in force).

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

Tests if `<token>` has the category code of a superscript token (`^` when normal `TEX` category codes are in force).

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

Tests if `<token>` has the category code of a subscript token (`_` when normal `TEX` category codes are in force).

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

Tests if `<token>` has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_letter_p:N</code>	<code>*</code>	<code>\token_if_letter_p:N</code>	<code><token></code>
<code>\token_if_letter:NTF</code>	<code>*</code>	<code>\token_if_letter:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of a letter token.

<code>\token_if_other_p:N</code>	<code>*</code>	<code>\token_if_other_p:N</code>	<code><token></code>
<code>\token_if_other:NTF</code>	<code>*</code>	<code>\token_if_other:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of an “other” token.

<code>\token_if_active_p:N</code>	<code>*</code>	<code>\token_if_active_p:N</code>	<code><token></code>
<code>\token_if_active:NTF</code>	<code>*</code>	<code>\token_if_active:NTF</code>	<code><token> {\true code} {\false code}</code>

Tests if `<token>` has the category code of an active character.

<code>\token_if_eq_catcode_p:NN</code>	<code>*</code>	<code>\token_if_eq_catcode_p:NN</code>	<code><token₁₂</code>
<code>\token_if_eq_catcode:NNTF</code>	<code>*</code>	<code>\token_if_eq_catcode:NNTF</code>	<code><token₁₂</code>

Tests if the two `<tokens>` have the same category code.

<code>\token_if_eq_charcode_p:NN</code>	<code>*</code>	<code>\token_if_eq_charcode_p:NN</code>	<code><token₁₂</code>
<code>\token_if_eq_charcode:NNTF</code>	<code>*</code>	<code>\token_if_eq_charcode:NNTF</code>	<code><token₁₂</code>

Tests if the two `<tokens>` have the same character code.

<code>\token_if_eq_meaning_p:NN</code>	★	<code>\token_if_eq_meaning_p:NN</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$
<code>\token_if_eq_meaning:NNTF</code>	★	<code>\token_if_eq_meaning:NNTF</code>	$\langle token_1 \rangle$	$\langle token_2 \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

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

<code>\token_if_macro_p:N</code>	★	<code>\token_if_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_macro:NNTF</code>	★	<code>\token_if_macro:NNTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2011-05-23

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

<code>\token_if_cs_p:N</code>	★	<code>\token_if_cs_p:N</code>	$\langle token \rangle$
<code>\token_if_cs:NNTF</code>	★	<code>\token_if_cs:NNTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

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

<code>\token_if_expandable_p:N</code>	★	<code>\token_if_expandable_p:N</code>	$\langle token \rangle$
<code>\token_if_expandable:NNTF</code>	★	<code>\token_if_expandable:NNTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

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

<code>\token_if_long_macro_p:N</code>	★	<code>\token_if_long_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_long_macro:NNTF</code>	★	<code>\token_if_long_macro:NNTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2012-01-20

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

<code>\token_if_protected_macro_p:N</code>	★	<code>\token_if_protected_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_protected_macro:NNTF</code>	★	<code>\token_if_protected_macro:NNTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is a protected macro: a macro which is both protected and long will return logical **false**.

<code>\token_if_protected_long_macro_p:N</code>	★	<code>\token_if_protected_long_macro_p:N</code>	$\langle token \rangle$
<code>\token_if_protected_long_macro:NNTF</code>	★	<code>\token_if_protected_long_macro:NNTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2012-01-20

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

<code>\token_if_chardef_p:N</code>	★	<code>\token_if_chardef_p:N</code>	$\langle token \rangle$
<code>\token_if_chardef:NNTF</code>	★	<code>\token_if_chardef:NNTF</code>	$\langle token \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Updated: 2012-01-20

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

\TeX hackers note: Booleans, boxes and small integer constants are implemented as chardefs.

<code>\token_if_mathchardef_p:N</code>	★	<code>\token_if_mathchardef_p:N</code>	$\langle token \rangle$
<code>\token_if_mathchardef:NTF</code>	★	<code>\token_if_mathchardef:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

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

<code>\token_if_dim_register_p:N</code>	★	<code>\token_if_dim_register_p:N</code>	$\langle token \rangle$
<code>\token_if_dim_register:NTF</code>	★	<code>\token_if_dim_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

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

<code>\token_if_int_register_p:N</code>	★	<code>\token_if_int_register_p:N</code>	$\langle token \rangle$
<code>\token_if_int_register:NTF</code>	★	<code>\token_if_int_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

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

TeXhackers note: Constant integers may be implemented as integer registers, chardefs, or mathchardefs depending on their value.

<code>\token_if_muskip_register_p:N</code>	★	<code>\token_if_muskip_register_p:N</code>	$\langle token \rangle$
<code>\token_if_muskip_register:NTF</code>	★	<code>\token_if_muskip_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

New: 2012-02-15

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

<code>\token_if_skip_register_p:N</code>	★	<code>\token_if_skip_register_p:N</code>	$\langle token \rangle$
<code>\token_if_skip_register:NTF</code>	★	<code>\token_if_skip_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

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

<code>\token_if_toks_register_p:N</code>	★	<code>\token_if_toks_register_p:N</code>	$\langle token \rangle$
<code>\token_if_toks_register:NTF</code>	★	<code>\token_if_toks_register:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2012-01-20

Tests if the $\langle token \rangle$ is defined to be a toks register (not used by L^AT_EX3).

<code>\token_if_primitive_p:N</code>	★	<code>\token_if_primitive_p:N</code>	$\langle token \rangle$
<code>\token_if_primitive:NTF</code>	★	<code>\token_if_primitive:NTF</code>	$\langle token \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Updated: 2011-05-23

Tests if the $\langle token \rangle$ is an engine primitive.

6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the “peek” functions. The generic `\peek_after:Nw` is provided along with a family of predefined tests for common cases. As peeking ahead does *not* skip spaces the predefined tests include both a space-respecting and space-skipping version.

`\peek_after:Nw`

`\peek_after:Nw <function> <token>`

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>` will remain in the input stream as the next item after the `<function>`. The `<token>` here may be `\`, `{` or `}` (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

`\peek_gafter:Nw`

`\peek_gafter:Nw <function> <token>`

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>` will remain in the input stream as the next item after the `<function>`. The `<token>` here may be `\`, `{` or `}` (assuming normal T_EX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

`\l_peek_token`

Token set by `\peek_after:Nw` and available for testing as described above.

`\g_peek_token`

Token set by `\peek_gafter:Nw` and available for testing as described above.

`\peek_catcode:NTF`

`\peek_catcode: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>` will be left in the input stream after the `<true code>` or `<false code>` (as appropriate to the result of the test).

`\peek_catcode_ignore_spaces:NNTF`

`\peek_catcode_ignore_spaces:NNTF <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>` will be left in the input stream after the `<true code>` or `<false code>` (as appropriate to the result of the test).

\peek_catcode_remove:NTF

Updated: 2012-12-20

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

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>* will be removed from the input stream if the test is true. The function will then place 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

Updated: 2012-12-20

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

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>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

\peek_charcode:NTF

Updated: 2012-12-20

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

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>* will be 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

Updated: 2012-12-20

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

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>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

\peek_charcode_remove:NTF

Updated: 2012-12-20

`\peek_charcode_remove:NTF <test token> {<true code>} {<false code>}`

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>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<code>\peek_charcode_remove_ignore_spaces:NTF</code>	<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>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<code>\peek_meaning:NTF</code>	<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>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<code>\peek_meaning_ignore_spaces:NTF</code>	<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>* will be left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

<code>\peek_meaning_remove:NTF</code>	<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>* will be removed from the input stream if the test is true. The function will then place either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

<code>\peek_meaning_remove_ignore_spaces:NTF</code>	<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>* will be removed from the input stream if the test is true. The function will then place 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 $\langle token \rangle$ passed is not a macro then no decomposition can occur. In the later 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 will leave 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 }`

will leave `#1#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` will be left in the input stream

TeXhackers note: If the arg spec. contains the string `->`, then the `spec` function will produce 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 will leave 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 }`

will leave `x#1 y#2` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` will be left in the input stream

`\token_get_prefix_spec:N` ★ `\token_get_prefix_spec:N` $\langle token \rangle$

If the $\langle token \rangle$ is a macro, this function will leave 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 }`

will leave `\long` in the input stream. If the $\langle token \rangle$ is not a macro then `\scan_stop:` will be left in the input stream

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 `)`. After two expansions, `\int_eval:n` yields an *⟨integer denotation⟩* which is left in the input stream. This 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 calculates the result of dividing the first value by the second, rounding any remainder. 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.

<hr/> <code>\int_div_truncate:nn</code> ★ <hr/>	<code>\int_div_truncate:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-02-09	Evaluates the two $\langle integer \text{ expressions} \rangle$ as described earlier, then calculates the result of dividing the first value by the second, truncating any remainder. Note that division using / rounds the result. The result is left in the input stream as an $\langle integer \text{ denotation} \rangle$ after two expansions.

<hr/> <code>\int_max:nn</code> ★	<code>\int_max:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
<hr/> <code>\int_min:nn</code> ★	<code>\int_min:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-09-26	Evaluates the $\langle integer \text{ expressions} \rangle$ as described for <code>\int_eval:n</code> and leaves either the larger or smaller value in the input stream as an $\langle integer \text{ denotation} \rangle$ after two expansions.

<hr/> <code>\int_mod:nn</code> ★	<code>\int_mod:nn {\langle integer \rangle_1} {\langle integer \rangle_2}</code>
Updated: 2012-09-26	Evaluates the two $\langle integer \text{ expressions} \rangle$ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is left in the input stream as an $\langle integer \text{ denotation} \rangle$ after two expansions.

2 Creating and initialising integers

<hr/> <code>\int_new:N</code>	<code>\int_new:N \langle integer \rangle</code>
<hr/> <code>\int_new:c</code>	Creates a new $\langle integer \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle integer \rangle$ will initially be equal to 0.

<hr/> <code>\int_const:Nn</code>	<code>\int_const:Nn \langle integer \rangle {\langle integer \text{ expression} \rangle}</code>
<hr/> <code>\int_const:cn</code>	Creates a new constant $\langle integer \rangle$ or raises an error if the name is already taken. The value of the $\langle integer \rangle$ will be set globally to the $\langle integer \text{ expression} \rangle$.
Updated: 2011-10-22	

<hr/> <code>\int_zero:N</code>	<code>\int_zero:N \langle integer \rangle</code>
<hr/> <code>\int_zero:c</code>	Sets $\langle integer \rangle$ to 0.
<hr/> <code>\int_gzero:N</code>	
<hr/> <code>\int_gzero:c</code>	

<hr/> <code>\int_zero_new:N</code>	<code>\int_zero_new:N \langle integer \rangle</code>
<hr/> <code>\int_zero_new:c</code>	Ensures that the $\langle integer \rangle$ exists globally by applying <code>\int_new:N</code> if necessary, then applies <code>\int_(g)zero:N</code> to leave the $\langle integer \rangle$ set to zero.
<hr/> <code>\int_gzero_new:N</code>	
<hr/> <code>\int_gzero_new:c</code>	
New: 2011-12-13	

<hr/> <code>\int_set_eq:NN</code>	<code>\int_set_eq:NN \langle integer_1 \rangle \langle integer_2 \rangle</code>
<hr/> <code>\int_set_eq:(cN Nc cc)</code>	Sets the content of $\langle integer_1 \rangle$ equal to that of $\langle integer_2 \rangle$.
<hr/> <code>\int_gset_eq:NN</code>	
<hr/> <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>	
<code>\int_gadd:Nn</code>	Adds the result of the $\langle integer\ expression \rangle$ to the current content of the $\langle integer \rangle$.
<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>	
<code>\int_gdecr:N</code>	Decreases the value stored in $\langle integer \rangle$ by 1.
<code>\int_gdecr:c</code>	

<code>\int_incr:N</code>	<code>\int_incr:N</code> $\langle integer \rangle$
<code>\int_incr:c</code>	
<code>\int_gincr:N</code>	Increases the value stored in $\langle integer \rangle$ by 1.
<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>	
<code>\int_gset:Nn</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: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>	
<code>\int_gsub:Nn</code>	Subtracts the result of the $\langle integer\ expression \rangle$ from the current content of the $\langle integer \rangle$.
<code>\int_gsub:cn</code>	

Updated: 2011-10-22

4 Using integers

<code>\int_use:N</code>	★	<code>\int_use:N <integer></code>
-------------------------	---	---

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

Updated: 2011-10-22		
---------------------	--	--

Recovers the content of an *<integer>* and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where an *<integer>* is required (such as in the first and third arguments of `\int_compare:nNnTF`).

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

5 Integer expression conditionals

<code>\int_compare_p:nNn</code>	★	<code>\int_compare_p:nNn {<intexpr₁>} <relation> {<intexpr₂>}</code>
---------------------------------	---	--

<code>\int_compare:nNnTF</code>	★	<code>\int_compare:nNnTF {<intexpr₁>} <relation> {<intexpr₂>} {<true code>} {<false code>}</code>
---------------------------------	---	---

This function first evaluates each of the *<integer expressions>* as described for `\int_eval:n`. The two results are then compared using the *<relation>*:

Equal	=
Greater than	>
Less than	<

```

\int_compare_p:n ★ \int_compare_p:n
\int_compare:nTF ★ {
    <intexpr1> <relation1>
    ...
    <intexprN> <relationN>
    <intexprN+1>
}
\int_compare:nTF
{
    <intexpr1> <relation1>
    ...
    <intexprN> <relationN>
    <intexprN+1>
}
{<true code>} {<false code>}

```

Updated: 2013-01-13

This function evaluates the *<integer expressions>* as described for `\int_eval:n` and compares consecutive result using the corresponding *<relation>*, namely it compares *<intexpr₁>* and *<intexpr₂>* using the *<relation₁>*, then *<intexpr₂>* and *<intexpr₃>* using the *<relation₂>*, until finally comparing *<intexpr_N>* and *<intexpr_{N+1}>* using the *<relation_N>*. The test yields **true** if all comparisons are **true**. Each *<integer expression>* is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is **false**, then no other *<integer expression>* is evaluated and no other comparison is performed. The *<relations>* can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

<code>\int_case:nnTF</code> ★	<code>\int_case:nnTF {<test integer expression>}</code>
<small>New: 2013-07-24</small>	<code>{</code> <code> {<intexpr case₁>} {<code case₁>}</code> <code> {<intexpr case₂>} {<code case₂>}</code> <code> ...</code> <code> {<intexpr case_n>} {<code case_n>}</code> <code>}</code> <code>{<true code>}</code> <code>{<false code>}</code>

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. 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! }
```

will leave “Medium” in the input stream.

<code>\int_if_even_p:n</code> ★	<code>\int_if_odd_p:n {<integer expression>}</code>
<code>\int_if_even:nTF</code> ★	<code>\int_if_odd:nTF {<integer expression>}</code>
<code>\int_if_odd_p:n</code> ★	<code>{<true code>} {<false code>}</code>
<code>\int_if_odd:nTF</code> ★	

This function first evaluates the *<integer expression>* as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

6 Integer expression loops

<code>\int_do_until:nNnn</code> ☆	<code>\int_do_until:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</code>
-----------------------------------	---

Places the *<code>* in the input stream for \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>* will be inserted into the input stream again and a loop will occur until the *<relation>* is **true**.

<hr/> <code>\int_do_while:nNnn</code> ☆ <hr/>	<code>\int_do_while:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</code>
	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<hr/> <code>\int_until_do:nNnn</code> ☆ <hr/>	<code>\int_until_do:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\int_while_do:nNnn</code> ☆ <hr/>	<code>\int_while_do:nNnn {<intexpr₁>} <relation> {<intexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><integer expressions></i> as described for <code>\int_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .
<hr/> <code>\int_do_until:nn</code> ☆ <hr/>	<code>\int_do_until:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is false then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is true .
<hr/> <code>\int_do_while:nn</code> ☆ <hr/>	<code>\int_do_while:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<hr/> <code>\int_until_do:nn</code> ☆ <hr/>	<code>\int_until_do:nn {<integer,elation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\int_while_do:nn</code> ☆ <hr/>	<code>\int_while_do:nn {<integer relation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <i><integer relation></i> as described for <code>\int_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .

7 Integer step functions

`\int_step_function:nnnN` ☆

New: 2012-06-04
Updated: 2012-06-29

`\int_step_function:nnnN` { $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } $\langle function \rangle$

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. The $\langle function \rangle$ is then placed in front of each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). Thus $\langle function \rangle$ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

```
[I saw 1]   [I saw 2]   [I saw 3]   [I saw 4]   [I saw 5]
```

`\int_step_inline:nnnn`

New: 2012-06-04
Updated: 2012-06-29

`\int_step_inline:nnnn` { $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } { $\langle code \rangle$ }

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. The $\langle code \rangle$ is then placed in front of each $\langle value \rangle$ from the $\langle initial\ value \rangle$ to the $\langle final\ value \rangle$ in turn (using $\langle step \rangle$ between each $\langle value \rangle$). Thus the $\langle code \rangle$ should define a function of one argument ($\#1$).

`\int_step_variable:nnnNn`

New: 2012-06-04
Updated: 2012-06-29

`\int_step_variable:nnnNn`
{ $\langle initial\ value \rangle$ } { $\langle step \rangle$ } { $\langle final\ value \rangle$ } $\langle tl\ var \rangle$ { $\langle code \rangle$ }

This function first evaluates the $\langle initial\ value \rangle$, $\langle step \rangle$ and $\langle final\ value \rangle$, all of which should be integer expressions. The $\langle code \rangle$ is inserted into the input stream, with the $\langle tl\ var \rangle$ defined as the current $\langle value \rangle$. Thus the $\langle code \rangle$ should make use of the $\langle tl\ var \rangle$.

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` { $\langle integer\ expression \rangle$ }

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

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

`\int_to_Alph: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_alph:n { 1 }`

places a in the input stream,

`\int_to_alph:n { 26 }`

is represented as z and

`\int_to_alph:n { 27 }`

is converted to aa. For conversions using other alphabets, use `\int_convert_to_symbols:nnn` to define an alphabet-specific function. The basic `\int_to_alph:n` and `\int_to_Alph:n` functions should not be modified.

`\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 (which will often be 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_alph:n` function is defined as

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

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

Updated: 2011-09-17

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

<code>\int_to_hexadecimal:n</code> ★	<code>\int_to_hexadecimal:n {\langle integer expression \rangle}</code>
Updated: 2011-09-17	Calculates the value of the $\langle integer expression \rangle$ and places the hexadecimal (base 16) representation of the result in the input stream. Upper case letters are used for digits beyond 9.

<code>\int_to_octal:n</code> ★	<code>\int_to_octal:n {\langle integer expression \rangle}</code>
Updated: 2011-09-17	Calculates the value of the $\langle integer expression \rangle$ and places the octal (base 8) representation of the result in the input stream.

<code>\int_to_base:nn</code> ★	<code>\int_to_base:nn {\langle integer expression \rangle} {\langle base \rangle}</code>
Updated: 2011-09-17	Calculates the value of the $\langle integer expression \rangle$ and converts it into the appropriate representation in the $\langle base \rangle$; the later may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by the upper case letters from the English alphabet. The maximum $\langle base \rangle$ value is 36.

TeXhackers note: This is a generic version of `\int_to_binary:n`, *etc.*

<code>\int_to_roman:n</code> ★	<code>\int_to_roman:n {\langle integer expression \rangle}</code>
<code>\int_to_Roman:n</code> ★	Places the value of the $\langle integer expression \rangle$ in the input stream as Roman numerals, either lower case (<code>\int_to_roman:n</code>) or upper case (<code>\int_to_Roman:n</code>). The Roman numerals are letters with category code 11 (letter).
Updated: 2011-10-22	

9 Converting from other formats to integers

<code>\int_from_alph:n</code> ★	<code>\int_from_alph:n {\langle letters \rangle}</code>
	Converts the $\langle letters \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle letters \rangle$ are treated using the English alphabet only, with “a” equal to 1 through to “z” equal to 26. Either lower or upper case letters may be used. This is the inverse function of <code>\int_to_alph:n</code> .

<code>\int_from_binary:n</code> ★	<code>\int_from_binary:n {\langle binary number \rangle}</code>
	Converts the $\langle binary number \rangle$ into the integer (base 10) representation and leaves this in the input stream.

<code>\int_from_hexadecimal:n</code> ★	<code>\int_from_hexadecimal:n {\langle hexadecimal number \rangle}</code>
	Converts the $\langle hexadecimal number \rangle$ into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the $\langle hexadecimal number \rangle$ by upper or lower case letters.

<hr/> <code>\int_from_octal:n</code> ★ <hr/>	<code>\int_from_octal:n {\langle octal number \rangle}</code> Converts the $\langle octal number \rangle$ into the integer (base 10) representation and leaves this in the input stream.
<hr/> <code>\int_from_roman:n</code> ★ <hr/>	<code>\int_from_roman:n {\langle roman numeral \rangle}</code> Converts the $\langle roman numeral \rangle$ into the integer (base 10) representation and leaves this in the input stream. The $\langle roman numeral \rangle$ may be in upper or lower case; if the numeral is not valid then the resulting value will be -1 .
<hr/> <code>\int_from_base:nn</code> ★ <hr/>	<code>\int_from_base:nn {\langle number \rangle} {\langle base \rangle}</code> Converts the $\langle number \rangle$ in $\langle base \rangle$ into the appropriate value in base 10. The $\langle number \rangle$ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum $\langle base \rangle$ value is 36.

10 Viewing integers

<hr/> <code>\int_show:N</code> <code>\int_show:c</code> <hr/>	<code>\int_show:N \langle integer \rangle</code> Displays the value of the $\langle integer \rangle$ on the terminal.
<hr/> <code>\int_show:n</code> New: 2011-11-22 Updated: 2012-05-27 <hr/>	<code>\int_show:n \langle integer expression \rangle</code> Displays the result of evaluating the $\langle integer expression \rangle$ on the terminal.

11 Constant integers

`\c_minus_one`
`\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.

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

<code>\if_int_compare:w</code> ★	<code>\if_int_compare:w</code> $\langle integer_1 \rangle$ $\langle relation \rangle$ $\langle integer_2 \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code> Compare two integers using $\langle relation \rangle$, which must be one of =, < or > with category code 12. The <code>\else:</code> branch is optional.
----------------------------------	---

T_EXhackers note: These are both names for the T_EX primitive `\ifnum`.

<code>\if_case:w</code> ★	<code>\if_case:w</code> $\langle integer \rangle$ $\langle case_0 \rangle$ <code>\or:</code> ★ $\langle case_1 \rangle$ <code>\or:</code> ... <code>\else:</code> $\langle default \rangle$ <code>\fi:</code> Selects a case to execute based on the value of the $\langle integer \rangle$. The first case ($\langle case_0 \rangle$) is executed if $\langle integer \rangle$ is 0, the second ($\langle case_1 \rangle$) if the $\langle integer \rangle$ is 1, <i>etc.</i> The $\langle integer \rangle$ may be a literal, a constant or an integer expression (<i>e.g.</i> using <code>\int_eval:n</code>).
---------------------------	--

T_EXhackers note: These are the T_EX primitives `\ifcase` and `\or`.

<code>\if_int_odd:w</code> ★	<code>\if_int_odd:w</code> $\langle tokens \rangle$ $\langle optional\ space \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle true\ code \rangle$ <code>\fi:</code> Expands $\langle tokens \rangle$ until a non-numeric token or a space is found, and tests whether the resulting $\langle integer \rangle$ is odd. If so, $\langle true\ code \rangle$ is executed. The <code>\else:</code> branch is optional.
------------------------------	---

T_EXhackers note: This is the T_EX primitive `\ifodd`.

14 Internal functions

<code>__int_to_roman:w</code> ★	<code>__int_to_roman:w</code> $\langle integer \rangle$ $\langle space \rangle$ or $\langle non-expandable\ token \rangle$ Converts $\langle integer \rangle$ to it 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 <i>are</i> 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.
----------------------------------	---

T_EXhackers note: This is the T_EX primitive `\romannumeral` renamed.

<code>__int_value:w</code>	★	<code>__int_value:w</code>	$\langle integer \rangle$
		<code>__int_value:w</code>	$\langle tokens \rangle$ $\langle optional\ space \rangle$

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

T_EXhackers note: This is the T_EX primitive `\number`.

<code>__int_eval:w</code>	★	<code>__int_eval:w</code>	$\langle intexpr \rangle$ <code>__int_eval_end:</code>
<code>__int_eval_end:</code>	★		

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.

T_EXhackers note: This is the ε -T_EX primitive `\numexpr`.

<code>__prg_compare_error:</code>	<code>__prg_compare_error:</code>
<code>__prg_compare_error:Nw</code>	<code>__prg_compare_error:Nw</code> $\langle token \rangle$

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

Part X

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:c`

`\dim_new:N` $\langle dimension \rangle$

Creates a new $\langle dimension \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle dimension \rangle$ will initially be equal to 0 pt.

`\dim_const:Nn`
`\dim_const:cn`

`\dim_const:Nn` $\langle dimension \rangle$ $\{ \langle dimension \text{ expression} \rangle \}$

Creates a new constant $\langle dimension \rangle$ or raises an error if the name is already taken. The value of the $\langle dimension \rangle$ will be set globally to the $\langle dimension \text{ expression} \rangle$.

New: 2012-03-05

`\dim_zero:N`
`\dim_zero:c`
`\dim_gzero:N`
`\dim_gzero:c`

`\dim_zero:N` $\langle dimension \rangle$

Sets $\langle dimension \rangle$ to 0 pt.

`\dim_zero_new:N`
`\dim_zero_new:c`
`\dim_gzero_new:N`
`\dim_gzero_new:c`

`\dim_zero_new:N` $\langle dimension \rangle$

Ensures that the $\langle dimension \rangle$ exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the $\langle dimension \rangle$ set to zero.

New: 2012-01-07

`\dim_if_exist_p:N` ★
`\dim_if_exist_p:c` ★
`\dim_if_exist:NTF` ★
`\dim_if_exist:cTF` ★

`\dim_if_exist_p:N` $\langle dimension \rangle$

`\dim_if_exist:NTF` $\langle dimension \rangle$ $\{ \langle true \text{ code} \rangle \} \{ \langle false \text{ code} \rangle \}$

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

New: 2012-03-03

2 Setting dim variables

<code>\dim_add:Nn</code>	<code>\dim_add:Nn <dimension> {<dimension expression>}</code>
<code>\dim_add:cn</code>	
<code>\dim_gadd:Nn</code>	Adds the result of the $\langle dimension\ expression \rangle$ to the current content of the $\langle dimension \rangle$.
<code>\dim_gadd:cn</code>	

Updated: 2011-10-22

<code>\dim_set:Nn</code>	<code>\dim_set:Nn <dimension> {<dimension expression>}</code>
<code>\dim_set:cn</code>	
<code>\dim_gset:Nn</code>	Sets $\langle dimension \rangle$ to the value of $\langle dimension\ expression \rangle$, which must evaluate to a length with units.
<code>\dim_gset:cn</code>	

Updated: 2011-10-22

<code>\dim_set_eq:NN</code>	<code>\dim_set_eq:NN <dimension₁> <dimension₂></code>
<code>\dim_set_eq:(cN Nc cc)</code>	
<code>\dim_gset_eq:NN</code>	Sets the content of $\langle dimension_1 \rangle$ equal to that of $\langle dimension_2 \rangle$.
<code>\dim_gset_eq:(cN Nc cc)</code>	

<code>\dim_sub:Nn</code>	<code>\dim_sub:Nn <dimension> {<dimension expression>}</code>
<code>\dim_sub:cn</code>	
<code>\dim_gsub:Nn</code>	Subtracts the result of the $\langle dimension\ expression \rangle$ from the current content of the $\langle dimension \rangle$.
<code>\dim_gsub:cn</code>	

Updated: 2011-10-22

3 Utilities for dimension calculations

<code>\dim_abs:n</code> ★	<code>\dim_abs:n {<dimexpr>}</code>
Updated: 2012-09-26	Converts the $\langle dimexpr \rangle$ to its absolute value, leaving the result in the input stream as a $\langle dimension\ denotation \rangle$.

<code>\dim_max:nn</code> ★	<code>\dim_max:nn {<dimexpr₁>} {<dimexpr₂>}</code>
<code>\dim_min:nn</code> ★	<code>\dim_min:nn {<dimexpr₁>} {<dimexpr₂>}</code>
New: 2012-09-09	
Updated: 2012-09-26	Evaluates the two $\langle dimension\ expressions \rangle$ and leaves either the maximum or minimum value in the input stream as appropriate, as a $\langle dimension\ denotation \rangle$.

<code>\dim_ratio:nn</code> ☆	<code>\dim_ratio:nn {<dimexpr₁>} {<dimexpr₂>}</code>
------------------------------	--

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
```

will display 327680/655360 on the terminal.

4 Dimension expression conditionals

<code>\dim_compare_p:nNn</code> ★	<code>\dim_compare_p:nNn {<dimexpr₁>} <relation> {<dimexpr₂>}</code>
<code>\dim_compare:nNnTF</code> ★	<code>\dim_compare:nNnTF</code> <code>{<dimexpr₁>} <relation> {<dimexpr₂>}</code> <code>{<true code>} {<false code>}</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:nnTF` ☆
 New: 2013-07-24

```
\dim_case:nnTF {<test dimension expression>}
{
  {<dimexpr case1>} {<code case1>}
  {<dimexpr case2>} {<code case2>}
  ...
  {<dimexpr casen>} {<code casen>}
}
{<true code>}
{<false code>}
```

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. 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! }
```

will leave “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>* will be inserted into the input stream again and a loop will occur 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>* will be inserted into the input stream again and a loop will occur until the *<relation>* is **false**.

<hr/> <code>\dim_until_do:nNnn</code> ☆ <hr/>	<code>\dim_until_do:nNnn {<dimexpr₁>} <relation> {<dimexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><dimension expressions></i> as described for <code>\dim_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\dim_while_do:nNnn</code> ☆ <hr/>	<code>\dim_while_do:nNnn {<dimexpr₁>} <relation> {<dimexpr₂>} {<code>}</code>
	Evaluates the relationship between the two <i><dimension expressions></i> as described for <code>\dim_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .
<hr/> <code>\dim_do_until:nn</code> ☆ <hr/>	<code>\dim_do_until:nn {<dimension relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is false then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is true .
<hr/> <code>\dim_do_while:nn</code> ☆ <hr/>	<code>\dim_do_while:nn {<dimension relation>} {<code>}</code>
Updated: 2013-01-13	Places the <i><code></i> in the input stream for T _E X to process, and then evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> . If the test is true then the <i><code></i> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<hr/> <code>\dim_until_do:nn</code> ☆ <hr/>	<code>\dim_until_do:nn {<dimension relation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is false . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is true .
<hr/> <code>\dim_while_do:nn</code> ☆ <hr/>	<code>\dim_while_do:nn {<dimension relation>} {<code>}</code>
Updated: 2013-01-13	Evaluates the <i><dimension relation></i> as described for <code>\dim_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .

6 Using dim expressions and variables

<hr/> <code>\dim_eval:n</code> ★ <hr/>	<code>\dim_eval:n {<dimension expression>}</code>
Updated: 2011-10-22	Evaluates the <i><dimension expression></i> , expanding any dimensions and token list variables within the <i><expression></i> to their content (without requiring <code>\dim_use:N/\tl_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a <i><dimension denotation></i> after two expansions. This will be expressed in points (pt), and will require suitable termination if used in a T _E X-style assignment as it is <i>not</i> an <i><internal dimension></i> .

<code>\dim_use:N</code>	★	<code>\dim_use:N</code> $\langle dimension \rangle$
-------------------------	---	---

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

Recovers the content of a $\langle dimension \rangle$ and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of `\dim_eval:n`).

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

7 Viewing dim variables

<code>\dim_show:N</code>	<code>\dim_show:N</code> $\langle dimension \rangle$
--------------------------	--

<code>\dim_show:c</code>	
--------------------------	--

Displays the value of the $\langle dimension \rangle$ on the terminal.

<code>\dim_show:n</code>	<code>\dim_show:n</code> $\langle dimension expression \rangle$
--------------------------	---

New: 2011-11-22	
Updated: 2012-05-27	

Displays the result of evaluating the $\langle dimension expression \rangle$ on the terminal.

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^AT_EX3-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^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 Creating and initialising skip variables

<code>\skip_new:N</code>	<code>\skip_new:N <skip></code>
<code>\skip_new:c</code>	Creates a new $\langle skip \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle skip \rangle$ will initially be equal to 0 pt.

<code>\skip_const:Nn</code>	<code>\skip_const:Nn <skip> {(skip expression)}</code>
<code>\skip_const:cn</code>	Creates a new constant $\langle skip \rangle$ or raises an error if the name is already taken. The value of the $\langle skip \rangle$ will be set globally to the $\langle skip expression \rangle$.

New: 2012-03-05

<code>\skip_zero:N</code>	<code>\skip_zero:N <skip></code>
<code>\skip_zero:c</code>	Sets $\langle skip \rangle$ to 0 pt.
<code>\skip_gzero:N</code>	
<code>\skip_gzero:c</code>	

<code>\skip_zero_new:N</code>	<code>\skip_zero_new:N <skip></code>
<code>\skip_zero_new:c</code>	Ensures that the $\langle skip \rangle$ exists globally by applying <code>\skip_new:N</code> if necessary, then applies <code>\skip_(g)zero:N</code> to leave the $\langle skip \rangle$ set to zero.
<code>\skip_gzero_new:N</code>	
<code>\skip_gzero_new:c</code>	

New: 2012-01-07

<code>\skip_if_exist_p:N</code> *	<code>\skip_if_exist_p:N <skip></code>
<code>\skip_if_exist_p:c</code> *	<code>\skip_if_exist:NNTF <skip> {(true code)} {(false code)}</code>
<code>\skip_if_exist:NNTF</code> *	Tests whether the $\langle skip \rangle$ is currently defined. This does not check that the $\langle skip \rangle$ really is a skip variable.
<code>\skip_if_exist:cNTF</code> *	

New: 2012-03-03

11 Setting skip variables

<code>\skip_add:Nn</code>	<code>\skip_add:Nn <skip> {(skip expression)}</code>
<code>\skip_add:cn</code>	Adds the result of the $\langle skip expression \rangle$ to the current content of the $\langle skip \rangle$.
<code>\skip_gadd:Nn</code>	
<code>\skip_gadd:cn</code>	

Updated: 2011-10-22

<code>\skip_set:Nn</code>	<code>\skip_set:Nn <skip> {(skip expression)}</code>
<code>\skip_set:cn</code>	Sets $\langle skip \rangle$ to the value of $\langle skip expression \rangle$, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).
<code>\skip_gset:Nn</code>	
<code>\skip_gset:cn</code>	

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 <skip1> <skip2>
```

Sets the content of $\langle skip_1 \rangle$ equal to that of $\langle skip_2 \rangle$.

```
\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn
```

```
\skip_sub:Nn <skip> {\skip expression}
```

Subtracts the result of the $\langle skip expression \rangle$ from the current content of the $\langle skip \rangle$.

Updated: 2011-10-22

12 Skip expression conditionals

```
\skip_if_eq_p:nn ★
\skip_if_eq:nnTF ★
```

```
\skip_if_eq_p:nn {\skipexpr1} {\skipexpr2}
\dim_compare:nTF
{\skipexpr1} {\skipexpr2}
{\true code} {\false code}
```

This function first evaluates each of the $\langle skip expressions \rangle$ as described for `\skip_eval:n`. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true.

```
\skip_if_finite_p:n ★
\skip_if_finite:nTF ★
```

New: 2012-03-05

```
\skip_if_finite_p:n {\skipexpr}
\skip_if_finite:nTF {\skipexpr} {\true code} {\false code}
```

Evaluates the $\langle skip expression \rangle$ as described for `\skip_eval:n`, and then tests if all of its components are finite.

13 Using skip expressions and variables

```
\skip_eval:n ★
```

Updated: 2011-10-22

```
\skip_eval:n {\skip expression}
```

Evaluates the $\langle skip expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring `\skip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle glue denotation \rangle$ after two expansions. This will be expressed in points (`pt`), and will require suitable termination if used in a T_EX-style assignment as it is *not* an $\langle internal glue \rangle$.

<hr/> <code>\skip_use:N</code> ★	<code>\skip_use:N</code> $\langle skip \rangle$
<hr/> <code>\skip_use:c</code> ★	Recovers the content of a $\langle skip \rangle$ and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of <code>\skip_eval:n</code>).

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

14 Viewing skip variables

<hr/> <code>\skip_show:N</code>	<code>\skip_show:N</code> $\langle skip \rangle$
<hr/> <code>\skip_show:c</code>	Displays the value of the $\langle skip \rangle$ on the terminal.
<hr/> <code>\skip_show:n</code>	<code>\skip_show:n</code> $\langle skip \text{ expression} \rangle$
<hr/> New: 2011-11-22 Updated: 2012-05-27	Displays the result of evaluating the $\langle skip \text{ expression} \rangle$ on the terminal.

15 Constant skips

<hr/> <code>\c_max_skip</code>	The maximum value that can be stored as a skip (equal to <code>\c_max_dim</code> in length), with no stretch nor shrink component.
<hr/> Updated: 2012-11-02	
<hr/> <code>\c_zero_skip</code>	A zero length as a skip, with no stretch nor shrink component.
<hr/> Updated: 2012-11-01	

16 Scratch skips

<hr/> <code>\l_tmpa_skip</code> <hr/> <code>\l_tmpb_skip</code>	Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_skip</code> <hr/> <code>\g_tmpb_skip</code>	Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

17 Inserting skips into the output

```
\skip_horizontal:N
\skip_horizontal:(c|n)
```

Updated: 2011-10-22

```
\skip_horizontal:N <skip>
\skip_horizontal:n {\<skipexpr>}
```

Inserts a horizontal $\langle skip \rangle$ into the current list.

T_EXhackers note: `\skip_horizontal:N` is the T_EX primitive `\hskip` renamed.

```
\skip_vertical:N
\skip_vertical:(c|n)
```

Updated: 2011-10-22

```
\skip_vertical:N <skip>
\skip_vertical:n {\<skipexpr>}
```

Inserts a vertical $\langle skip \rangle$ into the current list.

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 <muskip>
```

Creates a new $\langle muskip \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle muskip \rangle$ will initially be equal to 0 mu.

```
\muskip_const:Nn
\muskip_const:cn
```

New: 2012-03-05

```
\muskip_const:Nn <muskip> {\<muskip expression>}
```

Creates a new constant $\langle muskip \rangle$ or raises an error if the name is already taken. The value of the $\langle muskip \rangle$ will be set globally to the $\langle muskip expression \rangle$.

```
\muskip_zero:N
\muskip_zero:c
\muskip_gzero:N
\muskip_gzero:c
```

```
\skip_zero:N <muskip>
```

Sets $\langle muskip \rangle$ to 0 mu.

```
\muskip_zero_new:N
\muskip_zero_new:c
\muskip_gzero_new:N
\muskip_gzero_new:c
```

New: 2012-01-07

```
\muskip_zero_new:N <muskip>
```

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.

```
\muskip_if_exist_p:N ★
\muskip_if_exist_p:c ★
\muskip_if_exist:NTF ★
\muskip_if_exist:cTF ★
```

New: 2012-03-03

```
\muskip_if_exist_p:N <muskip>
\muskip_if_exist:NTF <muskip> {\<true code>} {\<false code>}
```

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

19 Setting muskip variables

<code>\muskip_add:Nn</code>	<code>\muskip_add:Nn <muskip> {<muskip expression>}</code>
<code>\muskip_add:cn</code>	
<code>\muskip_gadd:Nn</code>	Adds the result of the $\langle muskip expression \rangle$ to the current content of the $\langle muskip \rangle$.
<code>\muskip_gadd:cn</code>	
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 $\langle muskip \rangle$ to the value of $\langle muskip expression \rangle$, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).
<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 $\langle muskip_1 \rangle$ equal to that of $\langle muskip_2 \rangle$.
<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 $\langle muskip expression \rangle$ from the current content of the $\langle skip \rangle$.
<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 $\langle muskip expression \rangle$, expanding any skips and token list variables within the $\langle expression \rangle$ to their content (without requiring <code>\muskip_use:N/\tl_use:N</code>) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a $\langle muglue denotation \rangle$ after two expansions. This will be expressed in mu, and will require suitable termination if used in a TeX-style assignment as it is <i>not</i> an $\langle internal muglue \rangle$.
<code>\muskip_use:N</code> ★	<code>\muskip_use:N <muskip></code>
<code>\muskip_use:c</code> ★	
	Recovers the content of a $\langle skip \rangle$ and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. Can be omitted in places where a $\langle dimension \rangle$ is required (such as in the argument of <code>\muskip_eval:n</code>).

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

21 Viewing muskip variables

<hr/> <code>\muskip_show:N</code> <hr/>	<code>\muskip_show:N</code> $\langle muskip \rangle$
<code>\muskip_show:c</code> <hr/>	Displays the value of the $\langle muskip \rangle$ on the terminal.
<hr/> <code>\muskip_show:n</code> <hr/>	<code>\muskip_show:n</code> $\langle muskip\ expression \rangle$
New: 2011-11-22 Updated: 2012-05-27 <hr/>	Displays the result of evaluating the $\langle muskip\ expression \rangle$ on the terminal.

22 Constant muskips

<hr/> <code>\c_max_muskip</code> <hr/>	The maximum value that can be stored as a muskip, with no stretch nor shrink component.
<hr/> <code>\c_zero_muskip</code> <hr/>	A zero length as a muskip, with no stretch nor shrink component.

23 Scratch muskips

<hr/> <code>\l_tmpa_muskip</code> <hr/>	Scratch muskip for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_muskip</code> <hr/>	
<hr/> <code>\g_tmpa_muskip</code> <hr/>	Scratch muskip for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_muskip</code> <hr/>	

24 Primitive conditional

<hr/> <code>\if_dim:w</code> <hr/>	<code>\if_dim:w</code> $\langle dimen_1 \rangle$ $\langle relation \rangle$ $\langle dimen_2 \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false \rangle$ <code>\fi:</code>
	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_end:</code> ★	<code>_dim_eval:w</code> $\langle dimexpr \rangle$ <code>_dim_eval_end:</code>
---	--

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

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

<code>_dim_strip_bp:n</code> ★ <code>_dim_strip_pt:n</code> ★	<code>_dim_strip_bp:n</code> $\{ \langle dimension expression \rangle \}$ <code>_dim_strip_pt:n</code> $\{ \langle dimension expression \rangle \}$
--	--

New: 2011-11-11

Evaluates the $\langle dimension expression \rangle$, expanding any dimensions and token list variables within the $\langle expression \rangle$ to their content (without requiring `\dim_use:N/\tl_use:N`) and applying the standard mathematical rules. The magnitude of the result, expressed in big points (**bp**) or points (**pt**), will be left in the input stream with *no units*. If the decimal part of the magnitude is zero, this will be omitted.

If the $\{ \langle dimension expression \rangle \}$ contains additional units, these will be ignored, so for example

`_dim_strip_pt:n { 1 bp pt }`

will leave 1.00374 in the input stream (*i.e.* the magnitude of one “big point” when converted to points).

Part XI

The l3tl package

Token lists

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

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

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

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

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

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

```
{ Hello } ~ world
```

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

T_EXhackers note: When T_EX fetches an undelimited argument from the input stream, explicit character tokens with character code 32 (space) and category code 10 (space), which we here call “explicit space characters”, 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`.

When T_EX reads a character of category code 10 for the first time, it is converted to an explicit space character, with character code 32, regardless of the initial character code. “Funny” spaces with a different category code, can be produced using `\tl_to_lowercase:n` or `\tl_to_uppercase:n`. Explicit space characters are also produced as a result of `\token_to_str:N`, `\tl_to_str:n`, etc.

1 Creating and initialising token list variables

<hr/> <u>\tl_new:N</u> <u>\tl_new:c</u> <hr/>	<u>\tl_new:N</u> $\langle tl\ var \rangle$ Creates a new $\langle tl\ var \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle tl\ var \rangle$ will initially be empty.
<hr/> <u>\tl_const:Nn</u> <u>\tl_const:(Nx cn cx)</u> <hr/>	<u>\tl_const:Nn</u> $\langle tl\ var \rangle$ $\{ \langle token\ list \rangle \}$ Creates a new constant $\langle tl\ var \rangle$ or raises an error if the name is already taken. The value of the $\langle tl\ var \rangle$ will be set globally to the $\langle token\ list \rangle$.
<hr/> <u>\tl_clear:N</u> <u>\tl_clear:c</u> <u>\tl_gclear:N</u> <u>\tl_gclear:c</u> <hr/>	<u>\tl_clear:N</u> $\langle tl\ var \rangle$ Clears all entries from the $\langle tl\ var \rangle$.
<hr/> <u>\tl_clear_new:N</u> <u>\tl_clear_new:c</u> <u>\tl_gclear_new:N</u> <u>\tl_gclear_new:c</u> <hr/>	<u>\tl_clear_new:N</u> $\langle tl\ var \rangle$ Ensures that the $\langle tl\ var \rangle$ exists globally by applying <u>\tl_new:N</u> if necessary, then applies <u>\tl_(g)clear:N</u> to leave the $\langle tl\ var \rangle$ empty.
<hr/> <u>\tl_set_eq:NN</u> <u>\tl_set_eq:(cN Nc cc)</u> <u>\tl_gset_eq:NN</u> <u>\tl_gset_eq:(cN Nc cc)</u> <hr/>	<u>\tl_set_eq:NN</u> $\langle tl\ var_1 \rangle$ $\langle tl\ var_2 \rangle$ Sets the content of $\langle tl\ var_1 \rangle$ equal to that of $\langle tl\ var_2 \rangle$.
<hr/> <u>\tl_concat:NNN</u> <u>\tl_concat:ccc</u> <u>\tl_gconcat:NNN</u> <u>\tl_gconcat:ccc</u> <hr/>	<u>\tl_concat:NNN</u> $\langle tl\ var_1 \rangle$ $\langle tl\ var_2 \rangle$ $\langle tl\ var_3 \rangle$ Concatenates the content of $\langle tl\ var_2 \rangle$ and $\langle tl\ var_3 \rangle$ together and saves the result in $\langle tl\ var_1 \rangle$. The $\langle tl\ var_2 \rangle$ will be placed at the left side of the new token list.
<hr/> New: 2012-05-18 <hr/>	
<hr/> <u>\tl_if_exist_p:N</u> ★ <u>\tl_if_exist_p:c</u> ★ <u>\tl_if_exist:NTF</u> ★ <u>\tl_if_exist:cTF</u> ★ <hr/>	<u>\tl_if_exist_p:N</u> $\langle tl\ var \rangle$ <u>\tl_if_exist:NTF</u> $\langle tl\ var \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$ Tests whether the $\langle tl\ var \rangle$ is currently defined. This does not check that the $\langle tl\ var \rangle$ really is a token list variable.
<hr/> New: 2012-03-03 <hr/>	

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 $\langle tl\ var \rangle$ to contain $\langle tokens \rangle$, 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 $\langle tokens \rangle$ to the left side of the current content of $\langle tl\ var \rangle$.

<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 $\langle tokens \rangle$ to the right side of the current content of $\langle tl\ var \rangle$.

3 Modifying token list variables

<code>\tl_replace_once:Nnn</code>	<code>\tl_replace_once:Nnn <tl var> {<old tokens>} {<new tokens>}</code>
<code>\tl_replace_once:cnn</code>	
<code>\tl_greplace_once:Nnn</code>	
<code>\tl_greplace_once:cnn</code>	

Updated: 2011-08-11

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

<code>\tl_replace_all:Nnn</code>	<code>\tl_replace_all:Nnn <tl var> {<old tokens>} {<new tokens>}</code>
<code>\tl_replace_all:cnn</code>	
<code>\tl_greplace_all:Nnn</code>	
<code>\tl_greplace_all:cnn</code>	

Updated: 2011-08-11

Replaces all occurrences of $\langle old\ tokens \rangle$ in the $\langle tl\ var \rangle$ with $\langle new\ tokens \rangle$. $\langle Old\ tokens \rangle$ 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 $\langle old\ tokens \rangle$ may remain after the replacement (see `\tl_remove_all:Nn` for an example).

<code>\tl_remove_once:Nn</code>	<code>\tl_remove_once:Nn <tl var> {<tokens>}</code>
<code>\tl_remove_once:cn</code>	
<code>\tl_gremove_once:Nn</code>	
<code>\tl_gremove_once:cn</code>	

Updated: 2011-08-11

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

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

Updated: 2011-08-11

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

Removes all occurrences of $\langle tokens \rangle$ from the $\langle tl var \rangle$. $\langle Tokens \rangle$ 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 $\langle tokens \rangle$ may remain after the removal, for instance,

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

will result in \l_tmpa_tl containing `abcd`.

4 Reassigning token list category codes

```
\tl_set_rescan:Nnn
\tl_set_rescan:(Nno|Nnx|cnn|cno|cnx)
\tl_gset_rescan:Nnn
\tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)
```

Updated: 2011-12-18

```
\tl_set_rescan:Nnn <tl var> {<setup>} {<tokens>}
```

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. This allows the $\langle tl var \rangle$ to contain material with category codes other than those that apply when $\langle tokens \rangle$ are absorbed. See also $\backslash tl_rescan:nn$.

```
\tl_rescan:nn
```

Updated: 2011-12-18

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

Rescans $\langle tokens \rangle$ applying the category code régime specified in the $\langle setup \rangle$, and leaves the resulting tokens in the input stream. See also $\backslash tl_set_rescan:Nnn$.

5 Reassigning token list character codes

```
\tl_to_lowercase:n
```

Updated: 2012-09-08

```
\tl_to_lowercase:n {<tokens>}
```

Works through all of the $\langle tokens \rangle$, replacing each character token with the lower case equivalent as defined by $\backslash char_set_lccode:nn$. Characters with no defined lower case character code are left unchanged. This process does not alter the category code assigned to the $\langle tokens \rangle$.

T_EXhackers note: This is a wrapper around the T_EX primitive $\backslash lowercase$.

\tl_to_uppercase:nUpdated: 2012-09-08

\tl_to_uppercase:n $\{\langle tokens \rangle\}$

Works through all of the $\langle tokens \rangle$, replacing each character token with the upper case equivalent as defined by `\char_set_uccode:nn`. Characters with no defined upper case character code are left unchanged. This process does not alter the category code assigned to the $\langle tokens \rangle$.

TeXhackers note: This is a wrapper around the TeX primitive `\uppercase`.

6 Token list conditionals

\tl_if_blank_p:n ***\tl_if_blank_p:(V|o)** ***\tl_if_blank:nTF** ***\tl_if_blank:(V|o)TF** ***\tl_if_blank_p:n** $\{\langle token list \rangle\}$ **\tl_if_blank:nTF** $\{\langle token list \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

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.

\tl_if_empty_p:N ***\tl_if_empty_p:c** ***\tl_if_empty:NTF** ***\tl_if_empty:cTF** ***\tl_if_empty_p:N** $\langle tl var \rangle$ **\tl_if_empty:NTF** $\langle tl var \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

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

\tl_if_empty_p:n ***\tl_if_empty_p:(V|o)** ***\tl_if_empty:nTF** ***\tl_if_empty:(V|o)TF** ***\tl_if_empty_p:n** $\{\langle token list \rangle\}$ **\tl_if_empty:nTF** $\{\langle token list \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle token list \rangle$ is entirely empty (*i.e.* contains no tokens at all).

New: 2012-05-24

Updated: 2012-06-05

\tl_if_eq_p:NN ***\tl_if_eq_p:(Nc|cN|cc)** ***\tl_if_eq:NNTF** ***\tl_if_eq:(Nc|cN|cc)TF** ***\tl_if_eq_p:NN** $\{\langle tl var_1 \rangle\}$ $\{\langle tl var_2 \rangle\}$ **\tl_if_eq:NNTF** $\{\langle tl var_1 \rangle\}$ $\{\langle tl var_2 \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Compares the content of two $\langle token list variables \rangle$ and is logically **true** if the two contain the same list of tokens (*i.e.* identical in both the list of characters they contain and the category codes of those characters). Thus for example

```
\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**.

\tl_if_eq:nnTF**\tl_if_eq:nnTF** $\langle token list_1 \rangle$ $\{\langle token list_2 \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

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:cnTF</code>	<code>\tl_if_in:NnTF <tl var> {<token list>} {<true code>} {<false code>}</code>
--	--

Tests if the *<token list>* is found in the content of the *<tl var>*. The *<token list>* cannot contain the tokens `{`, `}` or `#` (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:(Vn on no)TF</code>	<code>\tl_if_in:nnTF {<token list₁>} {<token list₂>} {<true code>} {<false code>}</code>
--	--

Tests if *<token list₂>* is found inside *<token list₁>*. The *<token list₂>* cannot contain the tokens `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

<code>\tl_if_single_p:N</code> ★ <code>\tl_if_single_p:c</code> ★ <code>\tl_if_single:NTF</code> ★ <code>\tl_if_single:cTF</code> ★	<code>\tl_if_single_p:N <tl var></code> <code>\tl_if_single:NNTF <tl var> {<true code>} {<false code>}</code>
--	--

Updated: 2011-08-13

Tests if the content of the *<tl var>* consists of a single 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`.

<code>\tl_if_single_p:n</code> ★ <code>\tl_if_single:nTF</code> ★	<code>\tl_if_single_p:n {<token list>}</code> <code>\tl_if_single:nNTF {<token list>} {<true code>} {<false code>}</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`.

<code>\tl_case:NnnTF</code> ★ <code>\tl_case:cnnTF</code> ★	<code>\tl_case:NnTF <test token list variable></code> <code>{</code> <code> <token list variable case₁> {<code case₁>}</code> <code> <token list variable case₂> {<code case₂>}</code> <code> ...</code> <code> <token list variable case_n> {<code case_n>}</code> <code>}</code> <code>{<true code>}</code> <code>{<false code>}</code>
--	---

New: 2013-07-24

This function compares the *<test token list variable>* in turn with each of the *<token list variable cases>*. If the two are equal (as described for `\tl_if_eq:NNTF`) then the associated *<code>* is left in the input stream. 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.

7 Mapping to token lists

<hr/> <code>\tl_map_function:NN</code> ☆ <code>\tl_map_function:cN</code> ☆ <hr/> Updated: 2012-06-29	<code>\tl_map_function:NN</code> $\langle tl\ var \rangle$ $\langle function \rangle$ Applies $\langle function \rangle$ to every $\langle item \rangle$ in the $\langle tl\ var \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. This may be a number of tokens if the $\langle item \rangle$ was stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n-type arguments. See also <code>\tl_map_function:nN</code> .
<hr/> <code>\tl_map_function:nN</code> ☆ <hr/> Updated: 2012-06-29	<code>\tl_map_function:nN</code> $\langle token\ list \rangle$ $\langle function \rangle$ Applies $\langle function \rangle$ to every $\langle item \rangle$ in the $\langle token\ list \rangle$. The $\langle function \rangle$ will receive one argument for each iteration. This may be a number of tokens if the $\langle item \rangle$ was stored within braces. Hence the $\langle function \rangle$ should anticipate receiving n-type arguments. See also <code>\tl_map_function:NN</code> .
<hr/> <code>\tl_map_inline:Nn</code> <code>\tl_map_inline:cn</code> <hr/> Updated: 2012-06-29	<code>\tl_map_inline:Nn</code> $\langle tl\ var \rangle$ $\{ \langle inline\ function \rangle \}$ Applies the $\langle inline\ function \rangle$ to every $\langle item \rangle$ stored within the $\langle tl\ var \rangle$. The $\langle inline\ function \rangle$ should consist of code which will receive the $\langle item \rangle$ as #1. One in line mapping can be nested inside another. See also <code>\tl_map_function:NN</code> .
<hr/> <code>\tl_map_inline:nn</code> <hr/> Updated: 2012-06-29	<code>\tl_map_inline:nn</code> $\langle token\ list \rangle$ $\{ \langle inline\ function \rangle \}$ 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 will receive the $\langle item \rangle$ as #1. One in line mapping can be nested inside another. See also <code>\tl_map_function:nN</code> .
<hr/> <code>\tl_map_variable:NNn</code> <code>\tl_map_variable:cNn</code> <hr/> Updated: 2012-06-29	<code>\tl_map_variable:NNn</code> $\langle tl\ var \rangle$ $\langle variable \rangle$ $\{ \langle function \rangle \}$ 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 will receive the $\langle item \rangle$ stored in the $\langle variable \rangle$. One variable mapping can be nested inside another. See also <code>\tl_map_inline:Nn</code> .
<hr/> <code>\tl_map_variable:nNn</code> <hr/> Updated: 2012-06-29	<code>\tl_map_variable:nNn</code> $\langle token\ list \rangle$ $\langle variable \rangle$ $\{ \langle function \rangle \}$ 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 will receive the $\langle item \rangle$ stored in the $\langle variable \rangle$. One variable mapping can be nested inside another. See also <code>\tl_map_inline:nn</code> .

\tl_map_break: ☆

Updated: 2012-06-29

\tl_map_break:

Used to terminate a `\tl_map...` function before all entries in the *⟨token list variable⟩* have been processed. This will normally take 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 will lead 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 will depend 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 will normally take 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 will lead 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 will depend on the design of the mapping function.

8 Using token lists

\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.

<code>\tl_to_str:n</code>	★	<code>\tl_to_str:n {⟨tokens⟩}</code>
---------------------------	---	--------------------------------------

Converts the given $\langle tokens \rangle$ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This $\langle string \rangle$ is then left in the input stream. Note that this function requires only a single expansion.

T_EXhackers note: This is the ε -T_EX primitive `\detokenize`. Hence its argument *must* be given within braces.

<code>\tl_use:N</code>	★	<code>\tl_use:N ⟨tl var⟩</code>
<code>\tl_use:c</code>	★	

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

9 Working with the content of token lists

<code>\tl_count:n</code>	★	<code>\tl_count:n {⟨tokens⟩}</code>
<code>\tl_count:(V o)</code>	★	

New: 2012-05-13

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

<code>\tl_count:N</code>	★	<code>\tl_count:N ⟨tl var⟩</code>
<code>\tl_count:c</code>	★	

New: 2012-05-13

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

<code>\tl_reverse:n</code>	★	<code>\tl_reverse:n {⟨token list⟩}</code>
<code>\tl_reverse:(V o)</code>	★	

Updated: 2012-01-08

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

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

```
\tl_reverse:N
\tl_reverse:c
\tl_greverse:N
\tl_greverse:c
```

Updated: 2012-01-08

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

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

```
\tl_reverse_items:n *
```

New: 2012-01-08

```
\tl_reverse_items:n {\token list}
```

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

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

```
\tl_trim_spaces:n *
```

New: 2011-07-09
Updated: 2012-06-25

```
\tl_trim_spaces:n {\token list}
```

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.

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

```
\tl_trim_spaces:N
\tl_trim_spaces:c
\tl_gtrim_spaces:N
\tl_gtrim_spaces:c
```

New: 2011-07-09

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

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the content of the $\langle tl var \rangle$.

10 The first token from a token list

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

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

Updated: 2012-09-09

`\tl_head:n { abc }`

and

`\tl_head:n { ~ abc }`

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

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

yields `␣ab`. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) will result 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 will not expand further when appearing in an x-type argument expansion.

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

Leaves in the input stream the first *⟨item⟩* in the *⟨token list⟩*, discarding the rest of the *⟨token list⟩*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank *⟨token list⟩* (which consists only of space characters) will result 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 {⟨token list⟩}`

`\tl_tail:(n|V|v|f)` ★

Updated: 2012-09-01

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first *⟨item⟩* in the *⟨token list⟩*, 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 }`

will both leave `␣{bc}d` in the input stream. A blank *⟨token list⟩* (see `\tl_if_blank:nTF`) will result 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 will not expand further when appearing in an *x*-type argument expansion.

`\str_head:n` ★ `\str_head:n {⟨token list⟩}`

`\str_tail:n` ★ `\str_tail:n {⟨token list⟩}`

New: 2011-08-10

Converts the *⟨token list⟩* into a string, as described for `\tl_to_str:n`. The `\str_head:n` function then leaves the first character of this string in the input stream. The `\str_tail:n` function leaves all characters except the first in the input stream. The first character may be a space. If the *⟨token list⟩* argument is entirely empty, nothing is left in the input stream.

`\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 will always be **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 will always be **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 will always be **false**.

<code>\tl_if_head_is_group_p:n</code> ★	<code>\tl_if_head_is_group_p:n {⟨token list⟩}</code>
<code>\tl_if_head_is_group:nTF</code> ★	<code>\tl_if_head_is_group:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>
New: 2012-07-08	Tests if the first <i>⟨token⟩</i> in the <i>⟨token list⟩</i> is an explicit begin-group character (with category code 1 and any character code), in other words, if the <i>⟨token list⟩</i> starts with a brace group. In particular, the test is false if the <i>⟨token list⟩</i> starts with an implicit token such as <code>\c_group_begin_token</code> , or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

<code>\tl_if_head_is_N_type_p:n</code> ★	<code>\tl_if_head_is_N_type_p:n {⟨token list⟩}</code>
<code>\tl_if_head_is_N_type:nTF</code> ★	<code>\tl_if_head_is_N_type:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>
New: 2012-07-08	

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

<code>\tl_if_head_is_space_p:n</code> ★	<code>\tl_if_head_is_space_p:n {⟨token list⟩}</code>
<code>\tl_if_head_is_space:nTF</code> ★	<code>\tl_if_head_is_space:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}</code>
Updated: 2012-07-08	Tests if the first <i>⟨token⟩</i> in the <i>⟨token list⟩</i> is an explicit space character (explicit token with character code 12 and category code 10). In particular, the test is false if the <i>⟨token list⟩</i> starts with an implicit token such as <code>\c_space_token</code> , or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

11 Viewing token lists

<code>\tl_show:N</code>	<code>\tl_show:N ⟨tl var⟩</code>
<code>\tl_show:c</code>	Displays the content of the <i>⟨tl var⟩</i> on the terminal.
Updated: 2012-09-09	T_EXhackers note: This is similar to the T _E X primitive <code>\show</code> , wrapped to a fixed number of characters per line.

<code>\tl_show:n</code>	<code>\tl_show:n ⟨token list⟩</code>
Updated: 2012-09-09	Displays the <i>⟨token list⟩</i> on the terminal.
	T_EXhackers note: This is similar to the ε-T _E X primitive <code>\showtokens</code> , wrapped to a fixed number of characters per line.

12 Constant token lists

<u><code>\c_empty_tl</code></u>	Constant that is always empty.
<u><code>\c_job_name_tl</code></u>	Constant that gets the “job name” assigned when TeX starts.
<u>Updated: 2011-08-18</u>	TeXhackers note: This copies the contents of the primitive <code>\jobname</code> . It is a constant that is set by TeX and should not be overwritten by the package.
<u><code>\c_space_tl</code></u>	An explicit space character contained in a token list (compare this with <code>\c_space_token</code>). For use where an explicit space is required.

13 Scratch token lists

<u><code>\l_tmpa_tl</code></u> <u><code>\l_tmpb_tl</code></u>	Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<u><code>\g_tmpa_tl</code></u> <u><code>\g_tmpb_tl</code></u>	Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

14 Internal functions

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

Part XII

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` *⟨sequence⟩*

Creates a new *⟨sequence⟩* or raises an error if the name is already taken. The declaration is global. The *⟨sequence⟩* will initially contain no items.

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

`\seq_clear:N` *⟨sequence⟩*

Clears all items from the *⟨sequence⟩*.

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

`\seq_clear_new:N` *⟨sequence⟩*

Ensures that the *⟨sequence⟩* exists globally by applying `\seq_new:N` if necessary, then applies `\seq_(g)clear:N` to leave the *⟨sequence⟩* 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` *⟨sequence₁⟩* *⟨sequence₂⟩*

Sets the content of *⟨sequence₁⟩* equal to that of *⟨sequence₂⟩*.

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

`\seq_set_split:Nnn` *⟨sequence⟩* *{⟨delimiter⟩}* *{⟨token list⟩}*

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

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

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

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

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

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

```
\seq_if_exist_p:N <sequence>
\seq_if_exist:NTF <sequence> {\true code} {\false code}
```

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

New: 2012-03-03

2 Appending data to sequences

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

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

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

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

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

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

3 Recovering items from sequences

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

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

Updated: 2012-05-14

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

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

```
\seq_get_right:NN
\seq_get_right:cN
```

Updated: 2012-05-19

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

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 $\langle token list variable \rangle$ will contain the special marker `\q_no_value`.

`\seq_pop_left:NN`
`\seq_pop_left:cN`
 Updated: 2012-05-14

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

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

`\seq_gpop_left:NN`
`\seq_gpop_left:cN`
 Updated: 2012-05-14

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

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

`\seq_pop_right:NN`
`\seq_pop_right:cN`
 Updated: 2012-05-19

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

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

`\seq_gpop_right:NN`
`\seq_gpop_right:cN`
 Updated: 2012-05-19

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

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

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.

<code>\seq_pop_left:NNTF</code> <code>\seq_pop_left:cNTF</code>	<code>\seq_pop_left:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-14</p> <p style="margin: 0;">Updated: 2012-05-19</p>	<p>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>i.e.</i> removes the item from a $\langle sequence \rangle$. Both the $\langle sequence \rangle$ and the $\langle token list variable \rangle$ are assigned locally.</p>

<code>\seq_gpop_left:NNTF</code> <code>\seq_gpop_left:cNTF</code>	<code>\seq_gpop_left:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-14</p> <p style="margin: 0;">Updated: 2012-05-19</p>	<p>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>i.e.</i> 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.</p>

<code>\seq_pop_right:NNTF</code> <code>\seq_pop_right:cNTF</code>	<code>\seq_pop_right:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-19</p>	<p>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>i.e.</i> removes the item from a $\langle sequence \rangle$. Both the $\langle sequence \rangle$ and the $\langle token list variable \rangle$ are assigned locally.</p>

<code>\seq_gpop_right:NNTF</code> <code>\seq_gpop_right:cNTF</code>	<code>\seq_gpop_right:NNTF <sequence> <token list variable> {\true code} {\false code}</code>
<p style="margin: 0;">New: 2012-05-19</p>	<p>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>i.e.</i> 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.</p>

5 Modifying sequences

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

<code>\seq_remove_duplicates:N</code> <code>\seq_remove_duplicates:c</code> <code>\seq_gremove_duplicates:N</code> <code>\seq_gremove_duplicates:c</code>	<code>\seq_remove_duplicates:N <sequence></code>
	<p>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:nn(TF)</code>.</p>

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

<hr/>	
<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 <code><item></code> from the <code><sequence></code> . The <code><item></code> comparison takes place on a token basis, as for <code>\tl_if_eq:nn(TF)</code> .
<code>\seq_gremove_all:cn</code>	
<hr/>	

6 Sequence conditionals

<hr/>	
<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 <code><sequence></code> is empty (containing no items).
<code>\seq_if_empty:cTF</code> ★	
<hr/>	

<hr/>	
<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>	
<hr/>	

Tests if the `<item>` is present in the `<sequence>`.

7 Mapping to sequences

<hr/>	
<code>\seq_map_function:NN</code> ★	<code>\seq_map_function:NN <sequence> <function></code>
<code>\seq_map_function:cn</code> ★	Applies <code><function></code> to every <code><item></code> stored in the <code><sequence></code> . The <code><function></code> will receive one argument for each iteration. The <code><items></code> 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.
Updated: 2012-06-29	
<hr/>	

<hr/>	
<code>\seq_map_inline:Nn</code>	<code>\seq_map_inline:Nn <sequence> {<inline function>}</code>
<code>\seq_map_inline:cn</code>	Applies <code><inline function></code> to every <code><item></code> stored within the <code><sequence></code> . The <code><inline function></code> should consist of code which will receive the <code><item></code> as #1. One in line mapping can be nested inside another. The <code><items></code> are returned from left to right.
Updated: 2012-06-29	
<hr/>	

<hr/>	
<code>\seq_map_variable:NNn</code>	<code>\seq_map_variable:NNn <sequence> <tl var.> {<function using tl var.>}</code>
<code>\seq_map_variable:(Ncn cN ccn)</code>	
Updated: 2012-06-29	
<hr/>	

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. One variable mapping can be nested inside another. The `<items>` are returned from left to right.

\seq_map_break: ☆

Updated: 2012-06-29

\seq_map_break:

Used to terminate a `\seq_map...` function before all entries in the $\langle sequence \rangle$ have been processed. This will normally take 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 will lead 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 will depend on the design of the mapping function.

\seq_map_break:n ☆

Updated: 2012-06-29

\seq_map_break:n $\{\langle tokens \rangle\}$

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 will normally take 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 will lead 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 the $\langle tokens \rangle$ are inserted into the input stream. This will depend 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 $\langle sequence \rangle$ will include 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` $\langle seq\ var \rangle$ $\{\langle separator\ between\ two \rangle\}$
`\seq_use:cnnn` $\{\langle separator\ between\ more\ than\ two \rangle\}$ $\{\langle separator\ between\ final\ two \rangle\}$

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 will be 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~ }
```

will insert “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.

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

`\seq_use:Nn` ★
`\seq_use:cn` ★

New: 2013-05-26

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

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 will be 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~ }
```

will insert “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 $\langle items \rangle$ will not expand further when appearing in an x-type argument expansion.

9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data

functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

<code>\seq_get:NN</code> <code>\seq_get:cN</code> <hr/> Updated: 2012-05-14	<code>\seq_get:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ 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 $\langle token\ list\ variable \rangle$ will contain the special marker <code>\q_no_value</code> .
---	---

<code>\seq_pop:NN</code> <code>\seq_pop:cN</code> <hr/> Updated: 2012-05-14	<code>\seq_pop:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ 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 $\langle token\ list\ variable \rangle$ will contain the special marker <code>\q_no_value</code> .
---	---

<code>\seq_gpop:NN</code> <code>\seq_gpop:cN</code> <hr/> Updated: 2012-05-14	<code>\seq_gpop:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ 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 $\langle token\ list\ variable \rangle$ will contain the special marker <code>\q_no_value</code> .
---	--

<code>\seq_get:NNTF</code> <code>\seq_get:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19	<code>\seq_get:NNTF</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$ 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.
--	---

<code>\seq_pop:NNTF</code> <code>\seq_pop:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19	<code>\seq_pop:NNTF</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$ 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 top item from the $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$, <i>i.e.</i> removes the item from the $\langle sequence \rangle$. Both the $\langle sequence \rangle$ and the $\langle token\ list\ variable \rangle$ are assigned locally.
--	---

<code>\seq_gpop:NNTF</code> <code>\seq_gpop:cNTF</code> <hr/> New: 2012-05-14 Updated: 2012-05-19	<code>\seq_gpop:NNTF</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$ 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 top item from the $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$, <i>i.e.</i> removes the item from the $\langle sequence \rangle$. The $\langle sequence \rangle$ is modified globally, while the $\langle token\ list\ variable \rangle$ is assigned locally.
--	--

<code>\seq_push:Nn</code> <code>\seq_push:(NV Nv No Nx cn cV cv co cx)</code> <code>\seq_gpush:Nn</code> <code>\seq_gpush:(NV Nv No Nx cn cV cv co cx)</code>	<code>\seq_push:Nn</code> $\langle sequence \rangle$ $\{\langle item \rangle\}$ Adds the $\{\langle item \rangle\}$ to the top of the $\langle sequence \rangle$.
--	---

10 Constant and scratch sequences

`\c_empty_seq` Constant that is always empty.

New: 2012-07-02

`\l_tmpa_seq` Scratch sequences for local assignment. These are never used by the kernel code, and so
`\l_tmpb_seq` are safe for use with any L^AT_EX3-defined function. However, they may be overwritten by
 New: 2012-04-26 other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_seq` Scratch sequences for global assignment. These are never used by the kernel code, and
`\g_tmpb_seq` so are safe for use with any L^AT_EX3-defined function. However, they may be overwritten
 New: 2012-04-26 by other non-kernel code and so should only be used for short-term storage.

11 Viewing sequences

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

`\seq_show:c`

Displays the entries in the $\langle sequence \rangle$ in the terminal.

Updated: 2012-09-09

12 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 will be raised. The definition should always be set globally.

`__seq_push_item_def:n` `__seq_push_item_def:n` $\{\langle code \rangle\}$

`__seq_push_item_def:x`

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 XIII

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 }
```

will leave `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

<code>\clist_new:N</code>	<code>\clist_new:N <comma list></code>
---------------------------	--

<code>\clist_new:c</code>	
---------------------------	--

Creates a new *<comma list>* or raises an error if the name is already taken. The declaration is global. The *<comma list>* will initially contain no items.

<code>\clist_clear:N</code>	<code>\clist_clear:N <comma list></code>
-----------------------------	--

<code>\clist_clear:c</code>	
-----------------------------	--

<code>\clist_gclear:N</code>	
------------------------------	--

<code>\clist_gclear:c</code>	
------------------------------	--

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

<code>\clist_clear_new:N</code>	<code>\clist_clear_new:N <comma list></code>
---------------------------------	--

<code>\clist_clear_new:c</code>	
---------------------------------	--

<code>\clist_gclear_new:N</code>	
----------------------------------	--

<code>\clist_gclear_new:c</code>	
----------------------------------	--

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.

```

\clist_set_eq:NN
\clist_set_eq:(cN|Nc|cc)
\clist_gset_eq:NN
\clist_gset_eq:(cN|Nc|cc)

```

```
\clist_set_eq:NN <comma list1> <comma list2>
```

Sets the content of $\langle comma list_1 \rangle$ equal to that of $\langle comma list_2 \rangle$.

```

\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc

```

```
\clist_concat:NNN <comma list1> <comma list2> <comma list3>
```

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 $\langle comma list_2 \rangle$ will be placed at the left side of the new comma list.

```

\clist_if_exist_p:N ★
\clist_if_exist_p:c ★
\clist_if_exist:NTF ★
\clist_if_exist:cTF ★

```

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

```
\clist_if_exist:NTF <comma list> {\true code} {\false 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.

New: 2012-03-03

2 Adding data to comma lists

```

\clist_set:Nn
\clist_set:(NV|No|Nx|cn|cV|co|cx)
\clist_gset:Nn
\clist_gset:(NV|No|Nx|cn|cV|co|cx)

```

New: 2011-09-06

```
\clist_set:Nn <comma list> {\item1},...,\itemn}
```

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.

```

\clist_put_left:Nn
\clist_put_left:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_left:Nn
\clist_gput_left:(NV|No|Nx|cn|cV|co|cx)

```

Updated: 2011-09-05

```
\clist_put_left:Nn <comma list> {\item1},...,\itemn}
```

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:(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

```
\clist_put_right:Nn <comma list> {\item1},...,\itemn}
```

Appends the $\langle items \rangle$ to the right of the $\langle comma list \rangle$. 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.

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

Removes duplicate items from the *<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)`.

T_EXhackers 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 will not work if any of the items in the *<comma list>* contains `{`, `}`, or `#` (assuming the usual T_EX category codes apply).

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

Removes every occurrence of *<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

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

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

<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 nn nV no)TF</code>	

Updated: 2011-09-06

Tests if the $\langle item \rangle$ is present in the $\langle comma list \rangle$. In the case of an n -type $\langle comma list \rangle$, 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**.

T_EXhackers note: The $\langle item \rangle$ may not contain $\{$, $\}$, or $\#$ (assuming the usual T_EX 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 $\{a_{\square},_{\square}\{b\}_{\square},_{\square},\{c\},_{\square}\}$ then the arguments passed to the mapped function are ‘a’, ‘{b}_{\square}’, an empty argument, and ‘c’.

When the comma list is given as an N -type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using n -type comma lists.

<code>\clist_map_function:NN</code>	☆	<code>\clist_map_function:NN <comma list> <function></code>
<code>\clist_map_function:(cN nN)</code>	☆	

Updated: 2012-06-29

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

<code>\clist_map_inline:Nn</code>	<code>\clist_map_inline:Nn <comma list> {<inline function>}</code>
<code>\clist_map_inline:(cn nn)</code>	

Updated: 2012-06-29

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

<code>\clist_map_variable:Nn</code>	<code>\clist_map_variable:Nn <comma list> <tl var.> {<function using tl var.>}</code>
<code>\clist_map_variable:(cNn nNn)</code>	

Updated: 2012-06-29

Stores each entry in the *<comma list>* 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. One variable mapping can be nested inside another. The *<items>* are returned from left to right.

<code>\clist_map_break: ☆</code>	<code>\clist_map_break:</code>
----------------------------------	--------------------------------

Updated: 2012-06-29

Used to terminate a `\clist_map...` function before all entries in the *<comma list>* have been processed. This will normally take 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 will lead 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 further items are taken from the input stream. This will depend on the design of the mapping function.

`\clist_map_break:n` ☆

Updated: 2012-06-29

`\clist_map_break:n` { \langle tokens \rangle }

Used to terminate a `\clist_map...` function before all entries in the \langle comma list \rangle have been processed, inserting the \langle tokens \rangle after the mapping has ended. This will normally take 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 will lead 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 will depend on the design of the mapping function.

`\clist_count:N` ☆

`\clist_count:(c|n)` ☆

New: 2012-07-13

`\clist_count:N` \langle comma list \rangle

Leaves the number of items in the \langle comma list \rangle in the input stream as an \langle integer denotation \rangle . The total number of items in a \langle comma list \rangle will include those which are duplicates, *i.e.* every item in a \langle comma list \rangle is unique.

6 Using the content of comma lists directly

<code>\clist_use:Nnnn</code> ★ <code>\clist_use:cnnn</code> ★	<code>\clist_use:Nnnn <clist var> {<separator between two>}</code> <code>{<separator between more than two>} {<separator between final two>}</code>
--	--

New: 2013-05-26

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the appropriate $\langle\textit{separator}\rangle$ between the items. Namely, if the comma list has more than two items, the $\langle\textit{separator between more than two}\rangle$ is placed between each pair of items except the last, for which the $\langle\textit{separator between final two}\rangle$ is used. If the comma list has exactly two items, then they are placed in the input stream separated by the $\langle\textit{separator between two}\rangle$. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error will be 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~ }
```

will insert “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.

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

<code>\clist_use:Nn</code> ★ <code>\clist_use:cn</code> ★	<code>\clist_use:Nn <clist var> {<separator>}</code>
--	--

New: 2013-05-26

Places the contents of the $\langle\textit{clist var}\rangle$ in the input stream, with the $\langle\textit{separator}\rangle$ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error will be 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~ }
```

will insert “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 $\langle\textit{items}\rangle$ will 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.

<hr/> <u>\clist_get:NN</u> <u>\clist_get:cN</u> <hr/>	<hr/> <u>\clist_get:NN</u> $\langle comma list \rangle$ $\langle token list variable \rangle$ <hr/>
Updated: 2012-05-14	Stores the left-most item from a $\langle comma list \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle comma list \rangle$. The $\langle token list variable \rangle$ is assigned locally. If the $\langle comma list \rangle$ is empty the $\langle token list variable \rangle$ will contain the marker value <code>\q_no_value</code> .
<hr/> <u>\clist_get:NNTF</u> <u>\clist_get:cNTF</u> <hr/>	<hr/> <u>\clist_get:NNTF</u> $\langle comma list \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$ <hr/>
New: 2012-05-14	If the $\langle comma list \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle comma list \rangle$ is non-empty, stores the top item from the $\langle comma list \rangle$ in the $\langle token list variable \rangle$ without removing it from the $\langle comma list \rangle$. The $\langle token list variable \rangle$ is assigned locally.
<hr/> <u>\clist_pop:NN</u> <u>\clist_pop:cN</u> <hr/>	<hr/> <u>\clist_pop:NN</u> $\langle comma list \rangle$ $\langle token list variable \rangle$ <hr/>
Updated: 2011-09-06	Pops the left-most item from a $\langle comma list \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the comma list and stores it in the $\langle token list variable \rangle$. Both of the variables are assigned locally.
<hr/> <u>\clist_gpop:NN</u> <u>\clist_gpop:cN</u> <hr/>	<hr/> <u>\clist_gpop:NN</u> $\langle comma list \rangle$ $\langle token list variable \rangle$ <hr/>
	Pops the left-most item from a $\langle comma list \rangle$ into the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the comma list and stores it in the $\langle token list variable \rangle$. The $\langle comma list \rangle$ is modified globally, while the assignment of the $\langle token list variable \rangle$ is local.
<hr/> <u>\clist_pop:NNTF</u> <u>\clist_pop:cNTF</u> <hr/>	<hr/> <u>\clist_pop:NNTF</u> $\langle sequence \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$ <hr/>
New: 2012-05-14	If the $\langle comma list \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle comma list \rangle$ is non-empty, pops the top item from the $\langle comma list \rangle$ in the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the $\langle comma list \rangle$. Both the $\langle comma list \rangle$ and the $\langle token list variable \rangle$ are assigned locally.
<hr/> <u>\clist_gpop:NNTF</u> <u>\clist_gpop:cNTF</u> <hr/>	<hr/> <u>\clist_gpop:NNTF</u> $\langle comma list \rangle$ $\langle token list variable \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$ <hr/>
New: 2012-05-14	If the $\langle comma list \rangle$ is empty, leaves the $\langle false code \rangle$ in the input stream. The value of the $\langle token list variable \rangle$ is not defined in this case and should not be relied upon. If the $\langle comma list \rangle$ is non-empty, pops the top item from the $\langle comma list \rangle$ in the $\langle token list variable \rangle$, <i>i.e.</i> removes the item from the $\langle comma list \rangle$. The $\langle comma list \rangle$ is modified globally, while the $\langle token list variable \rangle$ is assigned locally.

<code>\clist_push:Nn</code>	<code>\clist_push:Nn <comma list> {<items>}</code>
<code>\clist_push:(NV No Nx cn cV co cx)</code>	
<code>\clist_gpush:Nn</code>	
<code>\clist_gpush:(NV No Nx cn cV co cx)</code>	

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

8 Viewing comma lists

<code>\clist_show:N</code>	<code>\clist_show:N <comma list></code>
<code>\clist_show:c</code>	Displays the entries in the <code><comma list></code> in the terminal.
Updated: 2012-09-09	

<code>\clist_show:n</code>	<code>\clist_show:n {<tokens>}</code>
Updated: 2012-09-09	Displays the entries in the comma list in the terminal.

9 Constant and scratch comma lists

<code>\c_empty_clist</code>	Constant that is always empty.
New: 2012-07-02	

<code>\l_tmpa_clist</code> <code>\l_tmpb_clist</code>	Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
New: 2011-09-06	

<code>\g_tmpa_clist</code> <code>\g_tmpb_clist</code>	Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any $\text{\LaTeX}3$ -defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
New: 2011-09-06	

Part XIV

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 will overwrite 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

 $\backslash\text{prop_new:N}$
 $\backslash\text{prop_new:c}$

 $\backslash\text{prop_new:N}$ $\langle property\ list \rangle$

Creates a new $\langle property\ list \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle property\ list \rangle$ will initially contain no entries.

 $\backslash\text{prop_clear:N}$
 $\backslash\text{prop_clear:c}$
 $\backslash\text{prop_gclear:N}$
 $\backslash\text{prop_gclear:c}$

 $\backslash\text{prop_clear:N}$ $\langle property\ list \rangle$

Clears all entries from the $\langle property\ list \rangle$.

 $\backslash\text{prop_clear_new:N}$
 $\backslash\text{prop_clear_new:c}$
 $\backslash\text{prop_gclear_new:N}$
 $\backslash\text{prop_gclear_new:c}$

 $\backslash\text{prop_clear_new:N}$ $\langle property\ list \rangle$

Ensures that the $\langle property\ list \rangle$ exists globally by applying $\backslash\text{prop_new:N}$ if necessary, then applies $\backslash\text{prop_gclear:N}$ to leave the list empty.

 $\backslash\text{prop_set_eq:NN}$
 $\backslash\text{prop_set_eq:(cN|Nc|cc)}$
 $\backslash\text{prop_gset_eq:NN}$
 $\backslash\text{prop_gset_eq:(cN|Nc|cc)}$

 $\backslash\text{prop_set_eq:NN}$ $\langle property\ list_1 \rangle$ $\langle property\ list_2 \rangle$

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> <p>If the <i><key></i> is present in the <i><property list></i> then no action is taken. If the <i><key></i> is not present in the <i><property list></i> then a new entry is added. Both the <i><key></i> and <i><value></i> may contain any <i><balanced text></i>. The <i><key></i> is stored after processing with <code>\tl_to_str:n</code>, meaning that category codes are ignored.</p>
--	---

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>* will contain 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> <p>Recovers the <i><value></i> stored with <i><key></i> from the <i><property list></i>, and places this in the <i><token list variable></i>. If the <i><key></i> is not found in the <i><property list></i> then the <i><token list variable></i> will contain the special marker <code>\q_no_value</code>. The <i><key></i> and <i><value></i> are then deleted from the property list. Both assignments are local. See also <code>\prop_pop:NnNTF</code>.</p>
--	---

Updated: 2011-08-18

<code>\prop_gpop:NnN</code> <code>\prop_gpop:(NoN cnN coN)</code>	<code>\prop_gpop:NnN <property list> {<key>} <tl var></code> <p>Recovers the <i><value></i> stored with <i><key></i> from the <i><property list></i>, and places this in the <i><token list variable></i>. If the <i><key></i> is not found in the <i><property list></i> then the <i><token list variable></i> will contain the special marker <code>\q_no_value</code>. The <i><key></i> and <i><value></i> are then deleted from the property list. The <i><property list></i> is modified globally, while the assignment of the <i><token list variable></i> is local. See also <code>\prop_gpop:NnNTF</code>.</p>
--	---

Updated: 2011-08-18

4 Modifying property lists

```
\prop_remove:Nn
\prop_remove:(NV|cn|cV)
\prop_gremove:Nn
\prop_gremove:(NV|cn|cV)
```

New: 2012-05-12

```
\prop_remove:Nn <property list> {<key>}
```

Removes the entry listed under $\langle key \rangle$ from the $\langle property list \rangle$. If the $\langle key \rangle$ is not found in the $\langle property list \rangle$ no change occurs, *i.e* there is no need to test for the existence of a key before deleting it.

5 Property list conditionals

```
\prop_if_exist_p:N *
\prop_if_exist_p:c *
\prop_if_exist:NTF *
\prop_if_exist:cTF *
```

New: 2012-03-03

```
\prop_if_exist_p:N <property list>
```

```
\prop_if_exist:NNTF <property list> {<true code>} {<false code>}
```

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

```
\prop_if_empty_p:N *
\prop_if_empty_p:c *
\prop_if_empty:NNTF *
\prop_if_empty:cTF *
```

```
\prop_if_empty_p:N <property list>
```

```
\prop_if_empty:NNTF <property list> {<true code>} {<false code>}
```

Tests if the $\langle property list \rangle$ is empty (containing no entries).

```
\prop_if_in_p:Nn *
\prop_if_in_p:(NV|No|cn|cV|co) *
\prop_if_in:NnNTF *
\prop_if_in:(NV|No|cn|cV|co)TF *
```

Updated: 2011-09-15

```
\prop_if_in:NnNTF <property list> {<key>} {<true code>} {<false code>}
```

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`.

T_EXhackers 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`
`\prop_get:(NVN|NoN|cnN|cVN|coN)TF`

Updated: 2012-05-19

`\prop_get:NnNTF` $\langle\textit{property list}\rangle$ $\{\langle\textit{key}\rangle\}$ $\langle\textit{token list variable}\rangle$
 $\{\langle\textit{true code}\rangle\}$ $\{\langle\textit{false code}\rangle\}$

If the $\langle\textit{key}\rangle$ is not present in the $\langle\textit{property list}\rangle$, leaves the $\langle\textit{false code}\rangle$ in the input stream. The value of the $\langle\textit{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\textit{key}\rangle$ is present in the $\langle\textit{property list}\rangle$, stores the corresponding $\langle\textit{value}\rangle$ in the $\langle\textit{token list variable}\rangle$ without removing it from the $\langle\textit{property list}\rangle$, then leaves the $\langle\textit{true code}\rangle$ in the input stream. The $\langle\textit{token list variable}\rangle$ is assigned locally.

`\prop_pop:NnNTF`
`\prop_pop:cnNTF`

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

`\prop_pop:NnNTF` $\langle\textit{property list}\rangle$ $\{\langle\textit{key}\rangle\}$ $\langle\textit{token list variable}\rangle$ $\{\langle\textit{true code}\rangle\}$
 $\{\langle\textit{false code}\rangle\}$

If the $\langle\textit{key}\rangle$ is not present in the $\langle\textit{property list}\rangle$, leaves the $\langle\textit{false code}\rangle$ in the input stream. The value of the $\langle\textit{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\textit{key}\rangle$ is present in the $\langle\textit{property list}\rangle$, pops the corresponding $\langle\textit{value}\rangle$ in the $\langle\textit{token list variable}\rangle$, *i.e.* removes the item from the $\langle\textit{property list}\rangle$. Both the $\langle\textit{property list}\rangle$ and the $\langle\textit{token list variable}\rangle$ are assigned locally.

`\prop_gpop:NnNTF`
`\prop_gpop:cnNTF`

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

`\prop_gpop:NnNTF` $\langle\textit{property list}\rangle$ $\{\langle\textit{key}\rangle\}$ $\langle\textit{token list variable}\rangle$ $\{\langle\textit{true code}\rangle\}$
 $\{\langle\textit{false code}\rangle\}$

If the $\langle\textit{key}\rangle$ is not present in the $\langle\textit{property list}\rangle$, leaves the $\langle\textit{false code}\rangle$ in the input stream. The value of the $\langle\textit{token list variable}\rangle$ is not defined in this case and should not be relied upon. If the $\langle\textit{key}\rangle$ is present in the $\langle\textit{property list}\rangle$, pops the corresponding $\langle\textit{value}\rangle$ in the $\langle\textit{token list variable}\rangle$, *i.e.* removes the item from the $\langle\textit{property list}\rangle$. The $\langle\textit{property list}\rangle$ is modified globally, while the $\langle\textit{token list variable}\rangle$ is assigned locally.

7 Mapping to property lists

`\prop_map_function:NN` ☆
`\prop_map_function:cn` ☆

Updated: 2013-01-08

`\prop_map_function:NN` $\langle\textit{property list}\rangle$ $\langle\textit{function}\rangle$

Applies $\langle\textit{function}\rangle$ to every $\langle\textit{entry}\rangle$ stored in the $\langle\textit{property list}\rangle$. The $\langle\textit{function}\rangle$ will receive two argument for each iteration: the $\langle\textit{key}\rangle$ and associated $\langle\textit{value}\rangle$. The order in which $\langle\textit{entries}\rangle$ are returned is not defined and should not be relied upon.

`\prop_map_inline:Nn`
`\prop_map_inline:cn`

Updated: 2013-01-08

`\prop_map_inline:Nn` $\langle\textit{property list}\rangle$ $\{\langle\textit{inline function}\rangle\}$

Applies $\langle\textit{inline function}\rangle$ to every $\langle\textit{entry}\rangle$ stored within the $\langle\textit{property list}\rangle$. The $\langle\textit{inline function}\rangle$ should consist of code which will receive the $\langle\textit{key}\rangle$ as #1 and the $\langle\textit{value}\rangle$ as #2. The order in which $\langle\textit{entries}\rangle$ 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 will normally take 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 will lead 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 will normally take 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 will lead to low level T_EX errors.

8 Viewing property lists

\prop_show:N**\prop_show:c**

Updated: 2012-09-09

\prop_show:N *⟨property list⟩*

Displays the entries in the *⟨property list⟩* in the terminal.

9 Scratch property lists

<code>\l_tmpa_prop</code>	Scratch property lists for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_prop</code>	
New: 2012-06-23	

<code>\g_tmpa_prop</code>	Scratch property lists for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\g_tmpb_prop</code>	
New: 2012-06-23	

10 Constants

<code>\c_empty_prop</code>	A permanently-empty property list used for internal comparisons.
----------------------------	--

11 Internal property list functions

<code>\s__prop</code>	The internal token used at the beginning of property lists. This is also used after each $\langle key \rangle$ (see <code>__prop_pair:wn</code>).
-----------------------	---

<code>__prop_pair:wn</code>	<code>__prop_pair:wn $\langle key \rangle$ \s__prop {$\langle item \rangle$}</code>
	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 will be raised. The definition should always be set globally.

<code>\l__prop_internal_tl</code>	Token list used to store new key–value pairs to be inserted by functions of the <code>\prop_put:Nnn</code> family.
-----------------------------------	--

<code>__prop_split:NnTF</code>	<code>__prop_split:NnTF $\langle property list \rangle$ {$\langle key \rangle$} {$\langle true code \rangle$} {$\langle false code \rangle$}</code>
Updated: 2013-01-08	Splits the $\langle property list \rangle$ at the $\langle key \rangle$, giving three token lists: the $\langle extract \rangle$ of $\langle property list \rangle$ before the $\langle key \rangle$, the $\langle value \rangle$ associated with the $\langle key \rangle$ and the $\langle extract \rangle$ of the $\langle property list \rangle$ after the $\langle value \rangle$. Both $\langle extracts \rangle$ retain the internal structure of a property list, and the concatenation of the two $\langle extracts \rangle$ is a property list. If the $\langle key \rangle$ is present in the $\langle property list \rangle$ then the $\langle true code \rangle$ is left in the input stream, with #1, #2, and #3 replaced by the first $\langle extract \rangle$, the $\langle value \rangle$, and the second extract. If the $\langle key \rangle$ is not present in the $\langle property list \rangle$ then the $\langle false code \rangle$ is left in the input stream, with no trailing material. Both $\langle true code \rangle$ and $\langle false code \rangle$ 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 $\langle true code \rangle$ for the three extracts from the property list. The $\langle key \rangle$ comparison takes place as described for <code>\str_if_eq:nn</code> .

Part XV

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 $\langle box \rangle$ or raises an error if the name is already taken. The declaration is global. The $\langle box \rangle$ will initially be void.

<code>\box_clear:N</code>	<code>\box_clear:N <box></code>
<code>\box_clear:c</code>	Clears the content of the $\langle box \rangle$ by setting the box equal to <code>\c_void_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 $\langle box \rangle$ exists globally by applying <code>\box_new:N</code> if necessary, then applies <code>\box_(g)clear:N</code> to leave the $\langle box \rangle$ empty.
<code>\box_gclear_new:N</code>	
<code>\box_gclear_new:c</code>	

<code>\box_set_eq:NN</code>	<code>\box_set_eq:NN <box₁> <box₂></code>
<code>\box_set_eq:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$.
<code>\box_gset_eq:NN</code>	
<code>\box_gset_eq:(cN Nc cc)</code>	

<code>\box_set_eq_clear:NN</code>	<code>\box_set_eq_clear:NN <box₁> <box₂></code>
<code>\box_set_eq_clear:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ within the current TeX group equal to that of $\langle box_2 \rangle$, then clears $\langle box_2 \rangle$ globally.

<code>\box_gset_eq_clear:NN</code>	<code>\box_gset_eq_clear:NN <box₁> <box₂></code>
<code>\box_gset_eq_clear:(cN Nc cc)</code>	Sets the content of $\langle box_1 \rangle$ equal to that of $\langle box_2 \rangle$, then clears $\langle box_2 \rangle$. These assignments are global.

<code>\box_if_exist_p:N</code> ★	<code>\box_if_exist_p:N</code> $\langle box \rangle$
<code>\box_if_exist_p:c</code> ★	<code>\box_if_exist:N</code> $\langle box \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<code>\box_if_exist:N</code> ★	Tests whether the $\langle box \rangle$ is currently defined. This does not check that the $\langle box \rangle$ really is a box.
<code>\box_if_exist:c</code> ★	

New: 2012-03-03

2 Using boxes

<code>\box_use:N</code>	<code>\box_use:N</code> $\langle box \rangle$
<code>\box_use:c</code>	Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting.

T_EXhackers note: This is the T_EX primitive `\copy`.

<code>\box_use_clear:N</code>	<code>\box_use_clear:N</code> $\langle box \rangle$
<code>\box_use_clear:c</code>	Inserts the current content of the $\langle box \rangle$ onto the current list for typesetting, then globally clears the content of the $\langle box \rangle$.

T_EXhackers note: This is the T_EX primitive `\box`.

<code>\box_move_right:nn</code>	<code>\box_move_right:nn</code> $\{\langle dimexpr \rangle\}$ $\{\langle box\ function \rangle\}$
<code>\box_move_left:nn</code>	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 <code>\box_use:N</code> $\langle box \rangle$ or a “raw” box specification such as <code>\vbox:n</code> $\{ xyz \}$.

<code>\box_move_up:nn</code>	<code>\box_move_up:nn</code> $\{\langle dimexpr \rangle\}$ $\{\langle box\ function \rangle\}$
<code>\box_move_down:nn</code>	This function operates in horizontal mode, and inserts the material specified by the $\langle box\ function \rangle$ such that its reference point is displaced vertical 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 <code>\box_use:N</code> $\langle box \rangle$ or a “raw” box specification such as <code>\vbox:n</code> $\{ xyz \}$.

3 Measuring and setting box dimensions

<code>\box_dp:N</code>	<code>\box_dp:N</code> $\langle box \rangle$
<code>\box_dp:c</code>	Calculates the depth (below the baseline) of the $\langle box \rangle$ in a form suitable for use in a $\langle dimension\ expression \rangle$.

T_EXhackers note: This is the T_EX primitive `\dp`.

<code>\box_ht:N</code>	<code>\box_ht:N <box></code>
<code>\box_ht:c</code>	Calculates the height (above the baseline) of the <code><box></code> in a form suitable for use in a <code><dimension expression></code> .

T_EXhackers note: This is the T_EX primitive `\ht`.

<code>\box_wd:N</code>	<code>\box_wd:N <box></code>
<code>\box_wd:c</code>	Calculates the width of the <code><box></code> in a form suitable for use in a <code><dimension expression></code> .

T_EXhackers note: This is the T_EX primitive `\wd`.

<code>\box_set_dp:Nn</code>	<code>\box_set_dp:Nn <box> {<dimension expression>}</code>
<code>\box_set_dp:cn</code>	Set the depth (below the baseline) of the <code><box></code> to the value of the <code>{<dimension expression>}</code> . This is a global assignment.
Updated: 2011-10-22	

<code>\box_set_ht:Nn</code>	<code>\box_set_ht:Nn <box> {<dimension expression>}</code>
<code>\box_set_ht:cn</code>	Set the height (above the baseline) of the <code><box></code> to the value of the <code>{<dimension expression>}</code> . This is a global assignment.
Updated: 2011-10-22	

<code>\box_set_wd:Nn</code>	<code>\box_set_wd:Nn <box> {<dimension expression>}</code>
<code>\box_set_wd:cn</code>	Set the width of the <code><box></code> to the value of the <code>{<dimension expression>}</code> . This is a global assignment.
Updated: 2011-10-22	

4 Box conditionals

<code>\box_if_empty_p:N</code> ★	<code>\box_if_empty_p:N <box></code>
<code>\box_if_empty_p:c</code> ★	<code>\box_if_empty:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_empty:NTF</code> ★	Tests if <code><box></code> is a empty (equal to <code>\c_empty_box</code>).
<code>\box_if_empty:cTF</code> ★	

<code>\box_if_horizontal_p:N</code> ★	<code>\box_if_horizontal_p:N <box></code>
<code>\box_if_horizontal_p:c</code> ★	<code>\box_if_horizontal:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_horizontal:NTF</code> ★	Tests if <code><box></code> is a horizontal box.
<code>\box_if_horizontal:cTF</code> ★	

<code>\box_if_vertical_p:N</code> ★	<code>\box_if_vertical_p:N <box></code>
<code>\box_if_vertical_p:c</code> ★	<code>\box_if_vertical:NTF <box> {<true code>} {<false code>}</code>
<code>\box_if_vertical:NTF</code> ★	Tests if <code><box></code> is a vertical box.
<code>\box_if_vertical:cTF</code> ★	

5 The last box inserted

<hr/> <code>\box_set_to_last:N</code> <hr/>	<code>\box_set_to_last:N</code> $\langle box \rangle$
<code>\box_set_to_last:c</code>	
<code>\box_gset_to_last:N</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 $\langle box \rangle$ will always be void as it is not possible to recover the last added item.
<code>\box_gset_to_last:c</code> <hr/>	

6 Constant boxes

<hr/> <code>\c_empty_box</code> <hr/>	This is a permanently empty box, which is neither set as horizontal nor vertical.
Updated: 2012-11-04 <hr/>	

7 Scratch boxes

<hr/> <code>\l_tmpa_box</code> <hr/>	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> <hr/>	
Updated: 2012-11-04 <hr/>	

<hr/> <code>\g_tmpa_box</code> <hr/>	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> <hr/>	

8 Viewing box contents

<hr/> <code>\box_show:N</code> <hr/>	<code>\box_show:N</code> $\langle box \rangle$
<code>\box_show:c</code>	
Updated: 2012-05-11 <hr/>	Shows full details of the content of the $\langle box \rangle$ in the terminal.
<hr/> <code>\box_show:Nnn</code> <hr/>	<code>\box_show:Nnn</code> $\langle box \rangle$ $\langle intexpr_1 \rangle$ $\langle intexpr_2 \rangle$
<code>\box_show:cnn</code> <hr/>	
New: 2012-05-11 <hr/>	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.
<hr/> <code>\box_log:N</code> <hr/>	<code>\box_log:N</code> $\langle box \rangle$
<code>\box_log:c</code> <hr/>	
New: 2012-05-11 <hr/>	Writes full details of the content of the $\langle box \rangle$ to the log.

<hr/> <code>\box_log:Nnn</code> <hr/>	<code>\box_show:Nnn</code> $\langle box \rangle$ $\langle intexpr_1 \rangle$ $\langle intexpr_2 \rangle$
<code>\box_log:cnn</code> <hr/>	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.
<code>New: 2012-05-11</code>	

9 Horizontal mode boxes

<hr/> <code>\hbox:n</code> <hr/>	<code>\hbox:n</code> $\{\langle contents \rangle\}$
	Typesets the $\langle contents \rangle$ into a horizontal box of natural width and then includes this box in the current list for typesetting.

T_EXhackers note: This is the T_EX primitive `\hbox`.

<hr/> <code>\hbox_to_wd:nn</code> <hr/>	<code>\hbox_to_wd:nn</code> $\{\langle dimexpr \rangle\}$ $\{\langle contents \rangle\}$
	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.

<hr/> <code>\hbox_to_zero:n</code> <hr/>	<code>\hbox_to_zero:n</code> $\{\langle contents \rangle\}$
	Typesets the $\langle contents \rangle$ into a horizontal box of zero width and then includes this box in the current list for typesetting.

<hr/> <code>\hbox_set:Nn</code> <hr/>	<code>\hbox_set:Nn</code> $\langle box \rangle$ $\{\langle contents \rangle\}$
<code>\hbox_set:cn</code> <hr/>	
<code>\hbox_gset:Nn</code> <hr/>	Typesets the $\langle contents \rangle$ at natural width and then stores the result inside the $\langle box \rangle$.
<code>\hbox_gset:cn</code> <hr/>	

<hr/> <code>\hbox_set_to_wd:Nnn</code> <hr/>	<code>\hbox_set_to_wd:Nnn</code> $\langle box \rangle$ $\{\langle dimexpr \rangle\}$ $\{\langle contents \rangle\}$
<code>\hbox_set_to_wd:cnn</code> <hr/>	
<code>\hbox_gset_to_wd:Nnn</code> <hr/>	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/>	

<hr/> <code>\hbox_overlap_right:n</code> <hr/>	<code>\hbox_overlap_right:n</code> $\{\langle contents \rangle\}$
	Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material will protrude to the right of the insertion point.

<hr/> <code>\hbox_overlap_left:n</code> <hr/>	<code>\hbox_overlap_left:n</code> $\{\langle contents \rangle\}$
	Typesets the $\langle contents \rangle$ into a horizontal box of zero width such that material will protrude to the left of the insertion point.

<hr/> <code>\hbox_set:Nw</code> <hr/>	<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>	

<hr/> <code>\hbox_unpack:N</code> <hr/>	<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.

T_EXhackers note: This is the T_EX primitive `\unhcopy`.

<hr/> <code>\hbox_unpack_clear:N</code> <hr/>	<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.

T_EXhackers note: This is the T_EX primitive `\unhbox`.

10 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 will have 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 will typically be non-zero.

<hr/> <code>\vbox:n</code> <hr/>	<code>\vbox:n {<contents>}</code>
Updated: 2011-12-18	Typesets the $\langle contents \rangle$ into a vertical box of natural height and includes this box in the current list for typesetting.

T_EXhackers note: This is the T_EX primitive `\vbox`.

<hr/> <code>\vbox_top:n</code> <hr/>	<code>\vbox_top:n {<contents>}</code>
Updated: 2011-12-18	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 will be equal to that of the <i>first</i> item added to the box.

T_EXhackers note: This is the T_EX primitive `\vtop`.

<hr/> <code>\vbox_to_ht:nn</code> <hr/>	<code>\vbox_to_ht:nn {<dimexpr>} {<contents>}</code>
Updated: 2011-12-18	Typesets the $\langle contents \rangle$ into a vertical box of height $\langle dimexpr \rangle$ and then includes this box in the current list for typesetting.
<hr/> <code>\vbox_to_zero:n</code> <hr/>	<code>\vbox_to_zero:n {<contents>}</code>
Updated: 2011-12-18	Typesets the $\langle contents \rangle$ into a vertical box of zero height and then includes this box in the current list for typesetting.
<hr/> <code>\vbox_set:Nn</code> <code>\vbox_set:cn</code> <code>\vbox_gset:Nn</code> <code>\vbox_gset:cn</code> <hr/>	<code>\vbox_set:Nn <box> {<contents>}</code> Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$.
Updated: 2011-12-18	
<hr/> <code>\vbox_set_top:Nn</code> <code>\vbox_set_top:cn</code> <code>\vbox_gset_top:Nn</code> <code>\vbox_gset_top:cn</code> <hr/>	<code>\vbox_set_top:Nn <box> {<contents>}</code> Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. The baseline of the box will be equal to that of the <i>first</i> item added to the box.
Updated: 2011-12-18	
<hr/> <code>\vbox_set_to_ht:Nnn</code> <code>\vbox_set_to_ht:cnn</code> <code>\vbox_gset_to_ht:Nnn</code> <code>\vbox_gset_to_ht:cnn</code> <hr/>	<code>\vbox_set_to_ht:Nnn <box> {<dimexpr>} {<contents>}</code> Typesets the $\langle contents \rangle$ to the height given by the $\langle dimexpr \rangle$ and then stores the result inside the $\langle box \rangle$.
Updated: 2011-12-18	
<hr/> <code>\vbox_set:Nw</code> <code>\vbox_set:cw</code> <code>\vbox_set_end:</code> <code>\vbox_gset:Nw</code> <code>\vbox_gset:cw</code> <code>\vbox_gset_end:</code> <hr/>	<code>\vbox_set:Nw <box> <contents> \vbox_set_end:</code> Typesets the $\langle contents \rangle$ at natural height and then stores the result inside the $\langle box \rangle$. In contrast to <code>\vbox_set:Nn</code> this function does not absorb the argument when finding the $\langle content \rangle$, and so can be used in circumstances where the $\langle content \rangle$ may not be a simple argument.
Updated: 2011-12-18	
<hr/> <code>\vbox_set_split_to_ht:NNn</code> <hr/>	<code>\vbox_set_split_to_ht:NNn <box₁₂</code>
Updated: 2011-10-22	Sets $\langle box_1 \rangle$ to contain material to the height given by the $\langle dimexpr \rangle$ by removing content from the top of $\langle box_2 \rangle$ (which must be a vertical box).

T_EXhackers note: This is the T_EX primitive `\vsplit`.

`\vbox_unpack:N`
`\vbox_unpack:c`

`\vbox_unpack:N` $\langle box \rangle$

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_clear:c`

`\vbox_unpack:N` $\langle box \rangle$

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`.

11 Primitive box conditionals

`\if_hbox:N` ★

`\if_hbox:N` $\langle box \rangle$
 $\langle true\ code \rangle$

`\else:`
 $\langle false\ code \rangle$

`\fi:`

Tests if $\langle box \rangle$ is a horizontal box.

T_EXhackers note: This is the T_EX primitive `\ifhbox`.

`\if_vbox:N` ★

`\if_vbox:N` $\langle box \rangle$
 $\langle true\ code \rangle$

`\else:`
 $\langle false\ code \rangle$

`\fi:`

Tests if $\langle box \rangle$ is a vertical box.

T_EXhackers note: This is the T_EX 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.

T_EXhackers note: This is the T_EX primitive `\ifvoid`.

Part XVI

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 $\langle coffin \rangle$ will initially be 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 TeX 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 TeX 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 TeX 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.

<hr/> <code>\hcoffin_set:Nw</code> <code>\hcoffin_set:cw</code> <code>\hcoffin_set_end:</code> <hr/> New: 2011-09-10	<code>\hcoffin_set:Nw <coffin> <material> \hcoffin_set_end:</code> 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.
<hr/> <code>\vcoffin_set:Nnn</code> <code>\vcoffin_set:cnn</code> <hr/> New: 2011-08-17 Updated: 2012-05-22	<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.
<hr/> <code>\vcoffin_set:Nnw</code> <code>\vcoffin_set:cnw</code> <code>\vcoffin_set_end:</code> <hr/> New: 2011-09-10 Updated: 2012-05-22	<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.
<hr/> <code>\coffin_set_horizontal_pole:Nnn</code> <code>\coffin_set_horizontal_pole:cnn</code> <hr/> New: 2012-07-20	<code>\coffin_set_horizontal_pole:Nnn <coffin> {<pole>} {<offset>}</code> Sets the $\langle pole \rangle$ to run horizontally through the $\langle coffin \rangle$. The $\langle pole \rangle$ will be located 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.
<hr/> <code>\coffin_set_vertical_pole:Nnn</code> <code>\coffin_set_vertical_pole:cnn</code> <hr/> New: 2012-07-20	<code>\coffin_set_vertical_pole:Nnn <coffin> {<pole>} {<offset>}</code> Sets the $\langle pole \rangle$ to run vertically through the $\langle coffin \rangle$. The $\langle pole \rangle$ will be located 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

<hr/>	<hr/>
<code>\coffin_attach:NnnNnnnn</code>	<code>\coffin_attach:NnnNnnnn</code>
<code>\coffin_attach:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code>\coffin_attach:(cnnNnnnn Nnnnnnn cnnnnnn)</code>
	<code>\coffin_1 \{ \coffin_1-pole_1 \} \{ \coffin_1-pole_2 \}</code>
	<code>\coffin_2 \{ \coffin_2-pole_1 \} \{ \coffin_2-pole_2 \}</code>
	<code>\{ \langle x-offset \rangle \} \{ \langle y-offset \rangle \}</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.

<hr/>	<hr/>
<code>\coffin_join:NnnNnnnn</code>	<code>\coffin_join:NnnNnnnn</code>
<code>\coffin_join:(cnnNnnnn Nnnnnnn cnnnnnn)</code>	<code>\coffin_join:(cnnNnnnn Nnnnnnn cnnnnnn)</code>
	<code>\coffin_1 \{ \coffin_1-pole_1 \} \{ \coffin_1-pole_2 \}</code>
	<code>\coffin_2 \{ \coffin_2-pole_1 \} \{ \coffin_2-pole_2 \}</code>
	<code>\{ \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 will cover 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.

<hr/>	<hr/>
<code>\coffin_typeset:Nnnnn</code>	<code>\coffin_typeset:Nnnnn \langle coffin \rangle \{ \langle pole_1 \rangle \} \{ \langle pole_2 \rangle \}</code>
<code>\coffin_typeset:cnnnn</code>	<code>\{ \langle x-offset \rangle \} \{ \langle y-offset \rangle \}</code>
<hr/>	<hr/>

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

<hr/>	<hr/>
<code>\coffin_dp:N</code>	<code>\coffin_dp:N \langle coffin \rangle</code>
<code>\coffin_dp:c</code>	
<hr/>	<hr/>

Calculates the depth (below the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

<hr/> <code>\coffin_ht:N</code> <hr/>	<code>\coffin_ht:N</code> $\langle coffin \rangle$
<code>\coffin_ht:c</code> <hr/>	Calculates the height (above the baseline) of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.
<hr/> <code>\coffin_wd:N</code> <hr/>	<code>\coffin_wd:N</code> $\langle coffin \rangle$
<code>\coffin_wd:c</code> <hr/>	Calculates the width of the $\langle coffin \rangle$ in a form suitable for use in a $\langle dimension expression \rangle$.

5 Coffin diagnostics

<hr/> <code>\coffin_display_handles:Nn</code> <hr/>	<code>\coffin_display_handles:Nn</code> $\langle coffin \rangle$ $\{ \langle colour \rangle \}$
<code>\coffin_display_handles:cn</code> <hr/>	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 $\langle handles \rangle$ will be labelled as part of this process: the locations of the $\langle handles \rangle$ and the labels are both printed in the $\langle colour \rangle$ specified.
Updated: 2011-09-02 <hr/>	

<hr/> <code>\coffin_mark_handle:Nnnn</code> <hr/>	<code>\coffin_mark_handle:Nnnn</code> $\langle coffin \rangle$ $\{ \langle pole_1 \rangle \}$ $\{ \langle pole_2 \rangle \}$ $\{ \langle colour \rangle \}$
<code>\coffin_mark_handle:cnnn</code> <hr/>	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 $\langle handle \rangle$ will be labelled as part of this process: the location of the $\langle handle \rangle$ and the label are both printed in the $\langle colour \rangle$ specified.
Updated: 2011-09-02 <hr/>	

<hr/> <code>\coffin_show_structure:N</code> <hr/>	<code>\coffin_show_structure:N</code> $\langle coffin \rangle$
<code>\coffin_show_structure:c</code> <hr/>	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.
Updated: 2012-09-09 <hr/>	
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.	

5.1 Constants and variables

<hr/> <code>\c_empty_coffin</code> <hr/>	A permanently empty coffin.
<hr/> <code>\l_tmpa_coffin</code> <hr/>	Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<code>\l_tmpb_coffin</code> <hr/>	
New: 2012-06-19	

Part XVII

The l3color package

Colour 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 Colour 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 will use the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin: ... \color_group_end: group`.

Part XVIII

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 will automatically be wrapped to the length available in the console. As a result, formatting is only needed where it will help 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 only those messages from the `submodule` to be filtered out.

```
\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 will be 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 will be raised if the *<message>* already exists.

<code>\msg_set:nnnn</code> <code>\msg_set:nnn</code> <code>\msg_gset:nnnn</code> <code>\msg_gset:nnn</code>	<code>\msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}</code> <p>Sets up the text for a <i><message></i> for a given <i><module></i>. The message will be defined to first give <i><text></i> and then <i><more text></i> if the user requests it. If no <i><more text></i> is available then a standard text is given instead. Within <i><text></i> and <i><more text></i> four parameters (#1 to #4) can be used: these will be supplied at the time the message is used.</p>
--	---

<code>\msg_if_exist_p:nn</code> ★ <code>\msg_if_exist:nnTF</code> ★ <div style="text-align: right; font-size: small;">New: 2012-03-03</div>	<code>\msg_if_exist_p:nn {<module>} {<message>}</code> <code>\msg_if_exist:nnTF {<module>} {<message>} {<true code>} {<false code>}</code> <p>Tests whether the <i><message></i> for the <i><module></i> is currently defined.</p>
---	--

2 Contextual information for messages

<code>\msg_line_context:</code> ☆	<code>\msg_line_context:</code> <p>Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is proceeded by the text on line.</p>
-----------------------------------	--

<code>\msg_line_number:</code> ★	<code>\msg_line_number:</code> <p>Prints the current line number when a message is given.</p>
----------------------------------	--

<code>\msg_fatal_text:n</code> ★	<code>\msg_fatal_text:n {<module>}</code> <p>Produces the standard text</p> <p style="text-align: center;">Fatal <i><module></i> error</p> <p>This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.</p>
----------------------------------	---

<code>\msg_critical_text:n</code> ★	<code>\msg_critical_text:n {<module>}</code> <p>Produces the standard text</p> <p style="text-align: center;">Critical <i><module></i> error</p> <p>This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.</p>
-------------------------------------	---

<code>\msg_error_text:n</code> ★	<code>\msg_error_text:n {<module>}</code> <p>Produces the standard text</p> <p style="text-align: center;"><i><module></i> error</p> <p>This function can be redefined to alter the language in which the message is given, using #1 as the name of the <i><module></i> to be included.</p>
----------------------------------	---

<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.

<code>\msg_info_text:n</code>	★	<code>\msg_info_text:n {<module>}</code>
-------------------------------	---	--

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.

<code>\msg_see_documentation_text:n</code>	★	<code>\msg_see_documentation_text:n {<module>}</code>
--	---	---

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 will be ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments will be converted to strings before being added to the message text: the `x`-type variants should be used to expand material.

<code>\msg_fatal:nnnnnn</code>	<code>\msg_fatal:nnnnnn {<module>} {<message>} {<arg one>}</code>
<code>\msg_fatal:(nnnnn nnnn nnn nn nnxxxx nnxxx nnxx nnx)</code>	<code>{<arg two>} {<arg three>} {<arg four>}</code>

Updated: 2012-08-11

Issues `<module> error <message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. After issuing a fatal error the `TEX` run will halt.

```
\msg_critical:nnnnnn  
\msg_critical:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a critical error, T_EX will stop reading the current input file. This may halt the T_EX run (if the current file is the main file) or may abort reading a sub-file.

T_EXhackers note: The T_EX `\endinput` primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

```
\msg_error:nnnnnn  
\msg_error:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The error will interrupt processing and issue the text at the terminal. After user input, the run will continue.

```
\msg_warning:nnnnnn  
\msg_warning:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues $\langle module \rangle$ warning $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The warning text will be added to the log file and the terminal, but the T_EX run will not be interrupted.

```
\msg_info:nnnnnn  
\msg_info:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues $\langle module \rangle$ information $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The information text will be added to the log file.

```
\msg_log:nnnnnn  
\msg_log:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues $\langle module \rangle$ information $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The information text will be added to the log file: the output is briefer than `\msg_info:nnnnnn`.

<code>\msg_none:nnnnnn</code> <code>\msg_none:(nnnnn nnnn nnn nn nnxxxx nnxxx nnxx nnx)</code>	<code>\msg_none:nnnnnn {<module>} {<message>} {<arg one>}</code> <code>{<arg two>} {<arg three>} {<arg four>}</code>
---	---

Updated: 2012-08-11

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

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 will raise 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 will raise errors 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>*.

<hr/> <code>\msg_redirect_module:nnn</code> <hr/>	<code>\msg_redirect_module:nnn {<module>} {<class one>} {<class two>}</code>
Updated: 2012-04-27	Redirects message of <i><class one></i> for <i><module></i> to act as though they were from <i><class two></i> . Messages of <i><class one></i> from sources other than <i><module></i> 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 <i><module></i> could be turned off with:

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

<hr/> <code>\msg_redirect_name:nnn</code> <hr/>	<code>\msg_redirect_name:nnn {<module>} {<message>} {<class>}</code>
Updated: 2012-04-27	Redirects a specific <i><message></i> from a specific <i><module></i> to act as a member of <i><class></i> 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.

<hr/> <code>\msg_interrupt:nnn</code> <hr/>	<code>\msg_interrupt:nnn {<first line>} {<text>} {<extra text>}</code>
New: 2012-06-28	Interrupts the TeX run, issuing a formatted message comprising <i><first line></i> and <i><text></i> laid out in the format

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! <first line>
!
! <text>
!.....
```

where the *<text>* will be wrapped to fit within the current line length. The user may then request more information, at which stage the *<extra text>* will be shown in the terminal in the format

```
|,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
| <extra text>
|.....
```

where the *<extra text>* will be wrapped within the current line length. Wrapping of both *<text>* and *<more text>* takes place using `\iow_wrap:nnn`; the documentation for the latter should be consulted for full details.

\msg_log:n

New: 2012-06-28

\msg_log:n {*text*}Writes to the log file with the *text* laid out in the format

```
.....  
. <text>  
.....
```

where the *text* will be 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

New: 2012-06-28

\msg_term:n {*text*}Writes to the terminal and log file with the *text* laid out in the format

```
*****  
* <text>  
*****
```

where the *text* will be 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:nnn

Updated: 2011-08-16

__msg_kernel_new:nnnn {*module*} {*message*} {*text*} {*more text*}

Creates a kernel *message* for a given *module*. The message will be 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 will be raised if the *message* already exists.

__msg_kernel_set:nnnn

__msg_kernel_set:nnn

__msg_kernel_set:nnnn {*module*} {*message*} {*text*} {*more text*}

Sets up the text for a kernel *message* for a given *module*. The message will be 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:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues kernel $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. After issuing a fatal error the T_EX run will halt. Cannot be redirected.

```
\_msg_kernel_error:nnnnnn
\_msg_kernel_error:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues kernel $\langle module \rangle$ error $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The error will stop processing and issue the text at the terminal. After user input, the run will continue. Cannot be redirected.

```
\_msg_kernel_warning:nnnnnn
\_msg_kernel_warning:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues kernel $\langle module \rangle$ warning $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The warning text will be added to the log file, but the T_EX run will not be interrupted.

```
\_msg_kernel_info:nnnnnn
\_msg_kernel_info:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)
```

Updated: 2012-08-11

Issues kernel $\langle module \rangle$ information $\langle message \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The information text will be 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:(nnnnn|nnnn|nnn|nn) ★
```

New: 2011-11-23

```
\_msg_kernel_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}
```

Issues an error, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. The resulting string must be shorter than a line, otherwise it will be cropped.

<code>_msg_expandable_error:n</code> ★	<code>_msg_expandable_error:n {⟨error message⟩}</code>
---	---

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.

<code>_msg_term:nnnnnn</code>	<code>_msg_term:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}</code>
<code>_msg_term:(nnnnnV nnnnn nnn nn)</code>	

Prints the *⟨message⟩* from *⟨module⟩* in the terminal without formatting. Used in messages which print complex variable contents completely.

<code>_msg_show_variable:Nnn</code>	<code>_msg_show_variable:Nnn ⟨variable⟩ {⟨type⟩} {⟨formatted content⟩}</code>
--------------------------------------	--

Updated: 2012-09-09

Displays the *⟨formatted content⟩* of the *⟨variable⟩* of *⟨type⟩* in the terminal. The *⟨formatted content⟩* will be processed as the first argument in a call to `\iow_wrap:nnnN`, hence `\`, `_` and other formatting sequences can be used. Once expanded and processed, the *⟨formatted content⟩* must either be empty or contain `>`; everything until the first `>` will be removed.

<code>_msg_show_variable:n</code>	<code>_msg_show_variable:n {⟨formatted text⟩}</code>
------------------------------------	---

Updated: 2012-09-09

Shows the *⟨formatted text⟩* on the terminal. After expansion, unless it is empty, the *⟨formatted text⟩* must contain `>`, and the part of *⟨formatted text⟩* before the first `>` is removed. Failure to do so causes low-level T_EX errors.

<code>_msg_show_item:n</code>	<code>_msg_show_item:n ⟨item⟩</code>
<code>_msg_show_item:nn</code>	<code>_msg_show_item:nn ⟨item-key⟩ ⟨item-value⟩</code>
<code>_msg_show_item_unbraced:nn</code>	

Updated: 2012-09-09

Auxiliary functions used within the argument of `_msg_show_variable:Nnn` 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.

Part XIX

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` will be 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 will be 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
}

```

will create a key called `\l_mymodule_tmp_tl`, and not one called `key`.

1 Creating keys

```

\keys_define:nn \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:
}

```

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.

```
.bool_set:N
.bool_set:c
.bool_gset:N
.bool_gset:c
```

Updated: 2013-07-08

$\langle key \rangle$.bool_set:N = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to $\langle value \rangle$ (which must be either **true** or **false**). If the variable does not exist, it will be created globally at the point that the key is set up.

```
.bool_set_inverse:N
.bool_set_inverse:c
.bool_gset_inverse:N
.bool_gset_inverse:c
```

New: 2011-08-28
Updated: 2013-07-08

$\langle key \rangle$.bool_set_inverse:N = $\langle boolean \rangle$

Defines $\langle key \rangle$ to set $\langle boolean \rangle$ to the logical inverse of $\langle value \rangle$ (which must be either **true** or **false**). If the $\langle boolean \rangle$ does not exist, it will be created globally at the point that the key is set up.

```
.choice:
```

$\langle key \rangle$.choice:

Sets $\langle key \rangle$ to act as a choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section [3](#).

```
.choices:nn
.choices:Vn
.choices:on
.choices:xn
```

New: 2011-08-21
Updated: 2013-07-10

$\langle key \rangle$.choices:nn $\langle choices \rangle$ $\langle code \rangle$

Sets $\langle key \rangle$ to act as a choice key, and defines a series $\langle choices \rangle$ which are implemented using the $\langle code \rangle$. Inside $\langle code \rangle$, $\backslash l_keys_choice_tl$ will be the name of the choice made, and $\backslash l_keys_choice_int$ will be the position of the choice in the list of $\langle choices \rangle$ (indexed from 1). Choices are discussed in detail in section [3](#).

```
.clist_set:N
.clist_set:c
.clist_gset:N
.clist_gset:c
```

New: 2011-09-11

$\langle key \rangle$.clist_set:N = $\langle comma list variable \rangle$

Defines $\langle key \rangle$ to set $\langle comma list variable \rangle$ to $\langle value \rangle$. Spaces around commas and empty items will be stripped. If the variable does not exist, it will be created globally at the point that the key is set up.

```
.code:n
```

Updated: 2013-07-10

$\langle key \rangle$.code:n = $\langle code \rangle$

Stores the $\langle code \rangle$ for execution when $\langle key \rangle$ is used. The $\langle code \rangle$ can include one parameter (**#1**), which will be the $\langle value \rangle$ given for the $\langle key \rangle$. The **x**-type variant will expand $\langle code \rangle$ at the point where the $\langle key \rangle$ is created.

<hr/> <code>.default:n</code> <hr/>	<code><key> .default:n = <default></code>
<code>.default:V</code>	
<code>.default:o</code>	Creates a <i><default></i> value for <i><key></i> , which is used if no value is given. This will be used
<code>.default:x</code> <hr/>	if only the key name is given, but not if a blank <i><value></i> is given:
Updated: 2013-07-09	<pre> \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 ' } </pre>
<hr/> <code>.dim_set:N</code> <hr/>	<code><key> .dim_set:N = <dimension></code>
<code>.dim_set:c</code>	
<code>.dim_gset:N</code>	Defines <i><key></i> to set <i><dimension></i> to <i><value></i> (which must a dimension expression). If the
<code>.dim_gset:c</code> <hr/>	variable does not exist, it will be created globally at the point that the key is set up.
<hr/> <code>.fp_set:N</code> <hr/>	<code><key> .fp_set:N = <floating point></code>
<code>.fp_set:c</code>	
<code>.fp_gset:N</code>	Defines <i><key></i> to set <i><floating point></i> to <i><value></i> (which must a floating point expression).
<code>.fp_gset:c</code> <hr/>	If the variable does not exist, it will be created globally at the point that the key is set up.
<hr/> <code>.groups:n</code> <hr/>	<code><key> .groups:n = <groups></code>
New: 2013-07-14	Defines <i><key></i> as belonging to the <i><groups></i> declared. Groups provide a “secondary axis”
	for selectively setting keys, and are described in Section 6 .
<hr/> <code>.initial:n</code> <hr/>	<code><key> .initial:n = <value></code>
<code>.initial:V</code>	
<code>.initial:o</code>	Initialises the <i><key></i> with the <i><value></i> , equivalent to
<code>.initial:x</code> <hr/>	<code>\keys_set:nn {<module>} { <key> = <value> }</code>
Updated: 2013-07-09	
<hr/> <code>.int_set:N</code> <hr/>	<code><key> .int_set:N = <integer></code>
<code>.int_set:c</code>	
<code>.int_gset:N</code>	Defines <i><key></i> to set <i><integer></i> to <i><value></i> (which must be an integer expression). If the
<code>.int_gset:c</code> <hr/>	variable does not exist, it will be created globally at the point that the key is set up.

<hr/> <code>.meta:n</code> <hr/>	<code><key> .meta:n = {<keyval list>}</code>
Updated: 2013-07-10	Makes <code><key></code> a meta-key, which will set <code><keyval list></code> in one go. If <code><key></code> is given with a value at the time the key is used, then the value will be passed through to the subsidiary <code><keys></code> for processing (as #1).
<hr/> <code>.meta:nn</code> <hr/>	<code><key> .meta:nn = {<path>} {<keyval list>}</code>
New: 2013-07-10	Makes <code><key></code> a meta-key, which will set <code><keyval list></code> in one go using the <code><path></code> in place of the current one. If <code><key></code> is given with a value at the time the key is used, then the value will be passed through to the subsidiary <code><keys></code> for processing (as #1).
<hr/> <code>.multichoice:</code> <hr/>	<code><key> .multichoice:</code>
New: 2011-08-21	Sets <code><key></code> to act as a multiple choice key. Each valid choice for <code><key></code> must then be created, as discussed in section 3.
<hr/> <code>.multichoices:nn</code> <code>.multichoices:Vn</code> <code>.multichoices:on</code> <code>.multichoices:xn</code> <hr/>	<code><key> .multichoices:nn <choices> <code></code>
New: 2011-08-21 Updated: 2013-07-10	Sets <code><key></code> to act as a multiple choice key, and defines a series <code><choices></code> which are implemented using the <code><code></code> . Inside <code><code></code> , <code>\l_keys_choice_tl</code> will be the name of the choice made, and <code>\l_keys_choice_int</code> will be the position of the choice in the list of <code><choices></code> (indexed from 1). Choices are discussed in detail in section 3.
<hr/> <code>.skip_set:N</code> <code>.skip_set:c</code> <code>.skip_gset:N</code> <code>.skip_gset:c</code> <hr/>	<code><key> .skip_set:N = <skip></code> Defines <code><key></code> to set <code><skip></code> to <code><value></code> (which must be a skip expression). If the variable does not exist, it will be created globally at the point that the key is set up.
<hr/> <code>.tl_set:N</code> <code>.tl_set:c</code> <code>.tl_gset:N</code> <code>.tl_gset:c</code> <hr/>	<code><key> .tl_set:N = <token list variable></code> Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> . If the variable does not exist, it will be created globally at the point that the key is set up.
<hr/> <code>.tl_set_x:N</code> <code>.tl_set_x:c</code> <code>.tl_gset_x:N</code> <code>.tl_gset_x:c</code> <hr/>	<code><key> .tl_set_x:N = <token list variable></code> Defines <code><key></code> to set <code><token list variable></code> to <code><value></code> , which will be subjected to an x-type expansion (<i>i.e.</i> using <code>\tl_set:Nx</code>). If the variable does not exist, it will be created globally at the point that the key is set up.
<hr/> <code>.value_forbidden:</code> <hr/>	<code><key> .value_forbidden:</code> Specifies that <code><key></code> cannot receive a <code><value></code> when used. If a <code><value></code> is given then an error will be issued.
<hr/> <code>.value_required:</code> <hr/>	<code><key> .value_required:</code> Specifies that <code><key></code> must receive a <code><value></code> when used. If a <code><value></code> is not given then an error will be issued.

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 will be 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)
```

```
\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 will be illustrated later.

```
\l_keys_key_tl
\l_keys_path_tl
\l_keys_value_tl
```

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` will look 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:(nVN|nvN|noN|nn|nV|nv|no)
```

```
\keys_set_known:nnN {<module>} {<keyval list>} <tl>
```

New: 2011-08-23

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 will be 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.

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` will attempt 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           ,
}
```

will assign `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made “active”, or by marking one or more groups to be ignored in key setting.

<code>\keys_set_filter:nnnN</code> <code>\keys_set_filter:(nnVN nnvN nnoN nnn nnV nnv nno)</code>	<code>\keys_set_filter:nnnN {<module>} {<groups>} {<keyval</code> <code>list>} <tl></code>
--	---

New: 2013-07-14

Active key filtering in an “opt-out” sense: keys assigned to any of the *<groups>* specified will be ignored. The *<groups>* are given as a comma-separated list. Unknown keys are not assigned to any group and will thus always be set. The key-value pairs for each key which is filtered out will be 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.

<code>\keys_set_groups:nnn</code>	<code>\keys_set_groups:nnn {<module>} {<groups>} {<keyval list>}</code>
<code>\keys_set_groups:(nnV nnv nno)</code>	

New: 2013-07-14

Activates key filtering in an “opt-in” sense: only keys assigned to one or more of the *<groups>* specified will be set. The *<groups>* are given as a comma-separated list. Unknown keys are not assigned to any group and will thus never be set.

7 Utility functions for keys

<code>\keys_if_exist_p:nn</code> ★	<code>\keys_if_exist_p:nn <module> <key></code>
<code>\keys_if_exist:nnTF</code> ★	<code>\keys_if_exist:nnTF <module> <key> {<true code>} {<false code>}</code>

Tests if the *<key>* exists for *<module>*, *i.e.* if any code has been defined for *<key>*.

<code>\keys_if_choice_exist_p:nnn</code> ★	<code>\keys_if_choice_exist_p:nnn <module> <key> <choice></code>
<code>\keys_if_choice_exist:nnnTF</code> ★	<code>\keys_if_choice_exist:nnnTF <module> <key> <choice> {<true code>} {<false code>}</code>

New: 2011-08-21

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.

<code>\keys_show:nn</code>	<code>\keys_show:nn {<module>} {<key>}</code>
----------------------------	---

Shows the function which is used to actually implement a *<key>* for a *<module>*.

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 will have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

`\keyval_parse:NNn`

Updated: 2011-09-08

`\keyval_parse:NNn` $\langle function_1 \rangle$ $\langle function_2 \rangle$ $\{ \langle key-value list \rangle \}$

Parses the $\langle key-value list \rangle$ into a series of $\langle keys \rangle$ and associated $\langle values \rangle$, or keys alone (if no $\langle value \rangle$ was given). $\langle function_1 \rangle$ should take one argument, while $\langle function_2 \rangle$ should absorb two arguments. After `\keyval_parse:NNn` has parsed the $\langle key-value list \rangle$, $\langle function_1 \rangle$ will be used to process keys given with no value and $\langle function_2 \rangle$ will be used to process keys given with a value. The order of the $\langle keys \rangle$ in the $\langle key-value list \rangle$ will be preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

will be converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the $\langle key \rangle$ and $\langle value \rangle$, then one *outer* set of braces is removed from the $\langle key \rangle$ and $\langle value \rangle$ as part of the processing.

Part XX

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 will attempt to locate them 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. Any active characters (as declared in `\l_char_active_seq`) will *not* be expanded, allowing the direct use of these in file names. Spaces are not allowed in file names.

1 File operation functions

<hr/> <code>\g_file_current_name_tl</code> <hr/>	Contains the name of the current \LaTeX file. This variable should not be modified: it is intended for information only. It will be equal to <code>\c_job_name_tl</code> at the start of a \LaTeX run and will be modified each time a file is read using <code>\file_input:n</code> .
<hr/> <code>\file_if_exist:nTF</code> <hr/> <div>Updated: 2012-02-10</div> <hr/>	<code>\file_if_exist:nTF {$\langle file\ name \rangle$} {$\langle true\ code \rangle$} {$\langle false\ code \rangle$}</code> Searches for $\langle file\ name \rangle$ using the current \TeX search path and the additional paths controlled by <code>\file_path_include:n</code> .
<hr/> <code>\file_add_path:nN</code> <hr/> <div>Updated: 2012-02-10</div> <hr/>	<code>\file_add_path:nN {$\langle file\ name \rangle$} $\langle tl\ var \rangle$</code> Searches for $\langle file\ name \rangle$ in the path as detailed for <code>\file_if_exist:nTF</code> , and if found sets the $\langle tl\ var \rangle$ the fully-qualified name of the file, <i>i.e.</i> the path and file name. If the file is not found then the $\langle tl\ var \rangle$ will contain the marker <code>\q_no_value</code> .
<hr/> <code>\file_input:n</code> <hr/> <div>Updated: 2012-02-17</div> <hr/>	<code>\file_input:n {$\langle file\ name \rangle$}</code> Searches for $\langle file\ name \rangle$ in the path as detailed for <code>\file_if_exist:nTF</code> , and if found reads in the file as additional \LaTeX source. All files read are recorded for information and the file name stack is updated by this function. An error will be raised if the file is not found.

<hr/> <code>\file_path_include:n</code> <hr/>	<code>\file_path_include:n {<path>}</code>
Updated: 2012-07-04	Adds $\langle path \rangle$ to the list of those used to search when reading files. The assignment is local. The $\langle path \rangle$ is processed in the same way as a $\langle file\ name \rangle$, <i>i.e.</i> , with <code>x</code> -type expansion except active characters. Spaces are not allowed in the $\langle path \rangle$.
<hr/> <code>\file_path_remove:n</code> <hr/>	<code>\file_path_remove:n {<path>}</code>
Updated: 2012-07-04	Removes $\langle path \rangle$ from the list of those used to search when reading files. The assignment is local. The $\langle path \rangle$ is processed in the same way as a $\langle file\ name \rangle$, <i>i.e.</i> , with <code>x</code> -type expansion except active characters. Spaces are not allowed in the $\langle path \rangle$.
<hr/> <code>\file_list:</code> <hr/>	<code>\file_list:</code>
	This function will list all files loaded using <code>\file_input:n</code> in the log file.

1.1 Input–output stream management

As \TeX is limited to 16 input streams and 16 output streams, direct use of the streams by the programmer is not supported in $\text{\LaTeX}3$. 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.

<hr/> <code>\ior_new:N</code> <hr/>	<code>\ior_new:N <stream></code>
<code>\ior_new:c</code>	<code>\iow_new:N <stream></code>
<code>\iow_new:N</code>	Globally reserves the name of the $\langle stream \rangle$, either for reading or for writing as appropriate. The $\langle stream \rangle$ is not opened until the appropriate <code>\..._open:Nn</code> function is used. Attempting to use a $\langle stream \rangle$ which has not been opened is an error, and the $\langle stream \rangle$ will behave as the corresponding <code>\c_term_...</code>
<code>\iow_new:c</code>	
New: 2011-09-26	
Updated: 2011-12-27	
<hr/> <code>\ior_open:Nn</code> <hr/>	<code>\ior_open:Nn <stream> {<file name>}</code>
<code>\ior_open:cn</code>	Opens $\langle file\ name \rangle$ for reading using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file\ name \rangle$ until a <code>\ior_close:N</code> instruction is given or the \TeX run ends.
Updated: 2012-02-10	
<hr/> <code>\ior_open:NnTF</code> <hr/>	<code>\ior_open:NnTF <stream> {<file name>} {<true code>} {<false code>}</code>
<code>\ior_open:cnTF</code>	Opens $\langle file\ name \rangle$ for reading using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file\ name \rangle$ until a <code>\ior_close:N</code> instruction is given or the \TeX run ends. The $\langle true\ code \rangle$ is then inserted into the input stream. If the file is not found, no error is raised and the $\langle false\ code \rangle$ is inserted into the input stream.
New: 2013-01-12	

<code>\iow_open:Nn</code>	<code>\iow_open:Nn <stream> {(file name)}</code>
<code>\iow_open:cn</code>	
Updated: 2012-02-09	

Opens $\langle file\ name \rangle$ for writing using $\langle stream \rangle$ as the control sequence for file access. If the $\langle stream \rangle$ was already open it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately and will remain allocated to $\langle file\ name \rangle$ until a `\iow_close:N` instruction is given or the T_EX run ends. Opening a file for writing will clear any existing content in the file (*i.e.* writing is *not* additive).

<code>\ior_close:N</code>	<code>\ior_close:N <stream></code>
<code>\ior_close:c</code>	<code>\iow_close:N <stream></code>
<code>\iow_close:N</code>	
<code>\iow_close:c</code>	
Updated: 2012-07-31	

Closes the $\langle stream \rangle$. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

<code>\ior_list_streams:</code>	<code>\ior_list_streams:</code>
<code>\iow_list_streams:</code>	<code>\iow_list_streams:</code>
Updated: 2012-09-09	

Displays a list of the file names associated with each open stream: intended for tracking down problems.

1.2 Reading from files

<code>\ior_get:NN</code>	<code>\ior_get:NN <stream> (token list variable)</code>
New: 2012-06-24	

Function that reads one or more lines (until an equal number of left and right braces are found) from the input $\langle stream \rangle$ 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 $\langle stream \rangle$ will be tokenized by T_EX according to the category codes in force when the function is used.

T_EXhackers note: This protected macro expands to the T_EX primitive `\read` along with the `to` keyword.

<code>\ior_get_str:NN</code>	<code>\ior_get_str:NN <stream> (token list variable)</code>
New: 2012-06-24	
Updated: 2012-07-31	

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).

T_EXhackers note: This protected macro is a wrapper around the ε -T_EX primitive `\readline`. However, the end-line character normally added by this primitive is not included in the result of `\ior_get_str:NN`.

<hr/>	<hr/>
<code>\ior_if_eof_p:N</code> ★	<code>\ior_if_eof_p:N</code> $\langle stream \rangle$
<code>\ior_if_eof:NTF</code> ★	<code>\ior_if_eof:NTF</code> $\langle stream \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
<hr/> Updated: 2012-02-10 <hr/>	Tests if the end of a $\langle stream \rangle$ has been reached during a reading operation. The test will also return a <code>true</code> value if the $\langle stream \rangle$ is not open.

2 Writing to files

<hr/>	<hr/>
<code>\iow_now:Nn</code>	<code>\iow_now:Nn</code> $\langle stream \rangle$ $\{\langle tokens \rangle\}$
<code>\iow_now:Nx</code>	This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ immediately (<i>i.e.</i> the write operation is called on expansion of <code>\iow_now:Nn</code>).
<hr/> Updated: 2012-06-05 <hr/>	

<hr/>	<hr/>
<code>\iow_log:n</code>	<code>\iow_log:n</code> $\{\langle tokens \rangle\}$
<code>\iow_log:x</code>	This function writes the given $\langle tokens \rangle$ to the log (transcript) file immediately: it is a dedicated version of <code>\iow_now:Nn</code> .
<hr/>	

<hr/>	<hr/>
<code>\iow_term:n</code>	<code>\iow_term:n</code> $\{\langle tokens \rangle\}$
<code>\iow_term:x</code>	This function writes the given $\langle tokens \rangle$ to the terminal file immediately: it is a dedicated version of <code>\iow_now:Nn</code> .
<hr/>	

<hr/>	<hr/>
<code>\iow_shipout:Nn</code>	<code>\iow_shipout:Nn</code> $\langle stream \rangle$ $\{\langle tokens \rangle\}$
<code>\iow_shipout:Nx</code>	This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (<i>i.e.</i> at shipout). The <code>x</code> -type variants expand the $\langle tokens \rangle$ at the point where the function is used but <i>not</i> when the resulting tokens are written to the $\langle stream \rangle$ (<i>cf.</i> <code>\iow_shipout_x:Nn</code>).
<hr/>	

<hr/>	<hr/>
<code>\iow_shipout_x:Nn</code>	<code>\iow_shipout_x:Nn</code> $\langle stream \rangle$ $\{\langle tokens \rangle\}$
<code>\iow_shipout_x:Nx</code>	This functions writes $\langle tokens \rangle$ to the specified $\langle stream \rangle$ when the current page is finalised (<i>i.e.</i> at shipout). The $\langle tokens \rangle$ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).
<hr/> Updated: 2012-09-08 <hr/>	

T_EXhackers note: This is a wrapper around the T_EX primitive `\write`.

<hr/>	<hr/>
<code>\iow_char:N</code> ★	<code>\iow_char:N</code> $\backslash \langle char \rangle$
<hr/>	Inserts $\langle char \rangle$ into the output stream. Useful when trying to write difficult characters such as <code>%</code> , <code>{</code> , <code>}</code> , <i>etc.</i> 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 $\langle tokens \rangle$ 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`).

2.1 Wrapping lines in output

\iow_wrap:nnnN $\{\langle text \rangle\} \{\langle run-on text \rangle\} \{\langle set up \rangle\} \langle function \rangle$

New: 2012-06-28

This function will wrap the $\langle text \rangle$ to a fixed number of characters per line. At the start of each line which is wrapped, the $\langle run-on text \rangle$ will be inserted. The line character count targeted will be the value of `\l_iow_line_count_int` minus the number of characters in the $\langle run-on text \rangle$. 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 message.

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 will typically be a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the $\langle function \rangle$) will consist of characters of category “other” (category code 12), with the exception of spaces which will have category “space” (category code 10). This means that the output will *not* expand further when written to a file.

T_EXhackers note: Internally, `\iow_wrap:nnnN` carries out an x-type expansion on the $\langle text \rangle$ to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` *could* be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the $\langle text \rangle$.

\iow_indent:n $\{\langle text \rangle\}$

New: 2011-09-21

In the context of `\iow_wrap:nnnN` (for instance in messages), indents $\langle text \rangle$ by four spaces. This function will 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.

<hr/> <code>\l_iow_line_count_int</code> <hr/>	
<hr/> New: 2012-06-24 <hr/>	
<hr/>	
<code>\c_catcode_other_space_tl</code> <hr/>	Token list containing one character with category code 12, (“other”), and character code 32 (space).
<hr/> New: 2011-09-05 <hr/>	
<hr/>	

2.2 Constant input–output streams

<hr/> <code>\c_term_iow</code> <hr/>	Constant input stream for reading from the terminal. Reading from this stream using <code>\ior_get:NN</code> or similar will result in a prompt from T _E X of the form <code><tl>=</code>
<hr/>	
<code>\c_log_iow</code> <code>\c_term_iow</code> <hr/>	Constant output streams for writing to the log and to the terminal (plus the log), respectively.

2.3 Primitive conditionals

<hr/> <code>\if_eof:w</code> ★ <hr/>	<pre> \if_eof:w <stream> <true code> \else: <false code> \fi: </pre> <p>Tests if the <i><stream></i> returns “end of file”, which is true for non-existent files. The <code>\else:</code> branch is optional.</p>
--------------------------------------	---

T_EXhackers note: This is the T_EX primitive `\ifeof`.

2.4 Internal file functions and variables

<hr/> <code>\l__file_internal_name_ior</code> <hr/>	Used to test for the existence of files when opening.
<hr/>	
<code>\l__file_internal_name_tl</code> <hr/>	Used to return the full name of a file for internal use.
<hr/>	

_file_name_sanitize:n

New: 2012-02-09

_file_name_sanitize:n {<name>} {<tokens>}

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 <sanitized name> is then inserted (in braces) after the <tokens>, which should further process the file name. If any spaces are found in the name after expansion, an error is raised.

2.5 Internal input–output functions

_ior_open:N

_ior_open:N

New: 2012-01-23

_ior_open:N <stream> {<file name>}

This function has identical syntax to the public version. However, it does not take precautions against active characters in the <file name>, and it does not attempt to add a <path> to the <file name>: it is therefore intended to be used by higher-level functions which have already fully expanded the <file name> and which need to perform multiple open or close operations. See for example the implementation of **\file_add_path:n**,

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 , and parentheses.
- Comparison operators: $x < y$, $x \leq y$, $x > y$, $x \neq y$ etc.
- Boolean logic: negation $!x$, conjunction $x \& \& y$, disjunction $x || y$, ternary operator $x ? y : z$.
- Exponentials: $\exp x$, $\ln x$, x^y .
- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$.

(not yet) Inverse trigonometric functions: $\operatorname{asin} x$, $\operatorname{acos} x$, $\operatorname{atan} x$, $\operatorname{acot} x$, $\operatorname{asec} x$, $\operatorname{acsc} x$.

(not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\operatorname{sech} x$, $\operatorname{csch} x$, and $\operatorname{asinh} x$, $\operatorname{acosh} x$, $\operatorname{atanh} x$, $\operatorname{acoth} x$, $\operatorname{asech} x$, $\operatorname{acsch} x$.

- Extrema: $\max(x, y, \dots)$, $\min(x, y, \dots)$, $\operatorname{abs}(x)$.
- Rounding functions: $\operatorname{round}(x, n)$ round to closest, $\operatorname{round0}(x, n)$ round towards zero, $\operatorname{round}\pm(x, n)$ round towards $\pm\infty$. And (not yet) modulo, and “quantize”.
- 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.

```
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnun } { m }
  { \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\calcnun { 2 pi * sin ( 2.3 ^ 5 ) }
```

1 Creating and initialising floating point variables

<hr/> <code>\fp_new:N</code> <code>\fp_new:c</code> <hr/>	<code>\fp_new:N <fp var></code>
Updated: 2012-05-08 <hr/>	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> will initially be +0.
<hr/> <code>\fp_const:Nn</code> <code>\fp_const:cn</code> <hr/>	<code>\fp_const:Nn <fp var> {<floating point expression>}</code>
Updated: 2012-05-08 <hr/>	Creates a new constant <i><fp var></i> or raises an error if the name is already taken. The <i><fp var></i> will be set globally equal to the result of evaluating the <i><floating point expression></i> .
<hr/> <code>\fp_zero:N</code> <code>\fp_zero:c</code> <code>\fp_gzero:N</code> <code>\fp_gzero:c</code> <hr/>	<code>\fp_zero:N <fp var></code>
Updated: 2012-05-08 <hr/>	Sets the <i><fp var></i> to +0.
<hr/> <code>\fp_zero_new:N</code> <code>\fp_zero_new:c</code> <code>\fp_gzero_new:N</code> <code>\fp_gzero_new:c</code> <hr/>	<code>\fp_zero_new:N <fp var></code>
Updated: 2012-05-08 <hr/>	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 zero.

2 Setting floating point variables

<hr/> <code>\fp_set:Nn</code> <code>\fp_set:cn</code> <code>\fp_gset:Nn</code> <code>\fp_gset:cn</code> <hr/>	<code>\fp_set:Nn <fp var> {<floating point expression>}</code>
Updated: 2012-05-08 <hr/>	Sets <i><fp var></i> equal to the result of computing the <i><floating point expression></i> .

```
\fp_set_eq:NN
\fp_set_eq:(cN|Nc|cc)
\fp_gset_eq:NN
\fp_gset_eq:(cN|Nc|cc)
```

Updated: 2012-05-08

```
\fp_set_eq:NN <fp var1> <fp var2>
```

Sets the floating point variable $\langle fp\ var_1 \rangle$ equal to the current value of $\langle fp\ var_2 \rangle$.

```
\fp_add:Nn
\fp_add:cn
\fp_gadd:Nn
\fp_gadd:cn
```

Updated: 2012-05-08

```
\fp_add:Nn <fp var> {\floating point expression}
```

Adds the result of computing the $\langle floating\ point\ expression \rangle$ to the $\langle fp\ var \rangle$.

```
\fp_sub:Nn
\fp_sub:cn
\fp_gsub:Nn
\fp_gsub:cn
```

Updated: 2012-05-08

```
\fp_sub:Nn <fp var> {\floating point expression}
```

Subtracts the result of computing the $\langle floating\ point\ expression \rangle$ from the $\langle fp\ var \rangle$.

3 Using floating point numbers

```
\fp_eval:n ★
```

New: 2012-05-08

Updated: 2012-07-08

```
\fp_eval:n {\floating point expression}
```

Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result as a decimal number with 16 significant figures and 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 `\fp_to_decimal:n`.

```
\fp_to_decimal:N ★
\fp_to_decimal:(c|n) ★
```

New: 2012-05-08

Updated: 2012-07-08

```
\fp_to_decimal:N <fp var>
```

```
\fp_to_decimal:n {\floating point expression}
```

Evaluates the $\langle floating\ point\ expression \rangle$ and expresses the result as a decimal number with 16 significant figures and 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.

```
\fp_to_dim:N ★
\fp_to_dim:(c|n) ★
```

Updated: 2012-07-08

```
\fp_to_dim:N <fp var>
```

```
\fp_to_dim:n {\floating point expression}
```

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 `\fp_to_decimal:n`, with an additional trailing **pt**. 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.

<code>\fp_to_int:N</code> ★	<code>\fp_to_int:N <fp var></code>
<code>\fp_to_int:(c n)</code> ★	<code>\fp_to_int:n {\floating point expression}</code>
Updated: 2012-07-08	Evaluates the <i><floating point expression></i> , and rounds the result to the closest integer, with ties rounded to an even integer. The result may be outside the range $[-2^{31} + 1, 2^{31} - 1]$ of valid TeX integers, triggering TeX errors if used in an integer expression. The values $\pm\infty$ and <code>nan</code> trigger an “invalid operation” exception.

<code>\fp_to_scientific:N</code> ★	<code>\fp_to_scientific:N <fp var></code>
<code>\fp_to_scientific:(c n)</code> ★	<code>\fp_to_scientific:n {\floating point expression}</code>
New: 2012-05-08 Updated: 2012-07-08	Evaluates the <i><floating point expression></i> and expresses the result in scientific notation with 16 significant figures:

<optional -> <digit> . <15 digits> e <optional sign> <exponent>

The leading *<digit>* is non-zero except in the case of ± 0 . The values $\pm\infty$ and `nan` trigger an “invalid operation” exception.

<code>\fp_to_tl:N</code> ★	<code>\fp_to_tl:N <fp var></code>
<code>\fp_to_tl:(c n)</code> ★	<code>\fp_to_tl:n {\floating point expression}</code>
Updated: 2012-07-08	Evaluates the <i><floating point expression></i> and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed (see <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 <code>-</code> . The special values ± 0 , $\pm\infty$ and <code>nan</code> are rendered as <code>0</code> , <code>-0</code> , <code>inf</code> , <code>-inf</code> , and <code>nan</code> respectively.

<code>\fp_use:N</code> ★	<code>\fp_use:N <fp var></code>
<code>\fp_use:c</code> ★	Inserts the value of the <i><fp var></i> into the input stream as a decimal number with 16 significant figures and 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 <code>nan</code> 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 <fp var></code>
<code>\fp_if_exist_p:c</code> ★	<code>\fp_if_exist:NTF <fp var> {\true code} {\false code}</code>
<code>\fp_if_exist:NTF</code> ★	Tests whether the <i><fp var></i> is currently defined. This does not check that the <i><fp var></i> really is a floating point variable.
<code>\fp_if_exist:cTF</code> ★	
Updated: 2012-05-08	

<code>\fp_compare_p:nNn</code> ★	<code>\fp_compare_p:nNn {<fpexpr₁>} <relation> {<fpexpr₂>}</code>
<code>\fp_compare_p:n</code> ★	<code>\fp_compare:nNnTF {<fpexpr₁>} <relation> {<fpexpr₂>} {<true code>} {<false code>}</code>
<code>\fp_compare:nNnTF</code> ★	<code>\fp_compare_p:n {<fpexpr₁>} <relation> {<fpexpr₂>}</code>
<code>\fp_compare:nTF</code> ★	<code>\fp_compare:nTF {<fpexpr₁>} <relation> {<fpexpr₂>} {<true code>} {<false code>}</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: $x \langle y, x=y, x \rangle y$, or x and y are not ordered. The latter case occurs exactly when one of the operands is **nan**, and this relations is denoted by the symbol **?**. The **nNn** functions support the $\langle relations \rangle$ **<**, **=**, **>**, and **?**. The **n** functions support as a $\langle relation \rangle$ any non-empty string of those four symbols, plus optional leading **!** (which negate the $\langle relation \rangle$), with the restriction that the $\langle relation \rangle$ may not start with **?**. Common choices of $\langle relation \rangle$ include **>=** (greater or equal), **!=** (not equal), **!?** (comparable). Note that a **nan** is distinct from any value, even another **nan**, hence $x = x$ is not true for a **nan**. Since a **nan** is not comparable to any floating point, to test if a value is **nan**, one can use the following, where 0 is an arbitrary floating point.

```
\fp_compare:nNnTF { <value> } ? { 0 }
{ } % <value> is nan
{ } % <value> is not nan
```

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 $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle floating\ point\ expressions \rangle$ as described for `\fp_compare:nNnTF`. If the test is **false** then the $\langle code \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation \rangle$ is **true**.

<code>\fp_do_while:nNnn</code> ☆	<code>\fp_do_while:nNnn {<fpexpr₁>} <relation> {<fpexpr₂>} {<code>}</code>
----------------------------------	--

New: 2012-08-16

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle floating\ point\ expressions \rangle$ as described for `\fp_compare:nNnTF`. If the test is **true** then the $\langle code \rangle$ will be inserted into the input stream again and a loop will occur until the $\langle relation \rangle$ 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 $\langle floating\ point\ expressions \rangle$ as described for `\fp_compare:nNnTF`, and then places the $\langle code \rangle$ in the input stream if the $\langle relation \rangle$ is **false**. After the $\langle code \rangle$ has been processed by T_EX the test will be repeated, and a loop will occur until the test is **true**.

<hr/> <code>\fp_while_do:nNnn</code> ☆ <hr/>	<code>\fp_while_do:nNnn {<fpexpr1>} <relation> {<fpexpr2>} {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .
<hr/> <code>\fp_do_until:nn</code> ☆ <hr/>	<code>\fp_do_until:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16	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> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is true .
<hr/> <code>\fp_do_while:nn</code> ☆ <hr/>	<code>\fp_do_while:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16	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> will be inserted into the input stream again and a loop will occur until the <i><relation></i> is false .
<hr/> <code>\fp_until_do:nn</code> ☆ <hr/>	<code>\fp_until_do:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16	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 will be repeated, and a loop will occur until the test is true .
<hr/> <code>\fp_while_do:nn</code> ☆ <hr/>	<code>\fp_while_do:nn { <fpexpr1> <relation> <fpexpr2> } {<code>}</code>
New: 2012-08-16	Evaluates the relationship between the two <i><floating point expressions></i> as described for <code>\fp_compare:nTF</code> , and then places the <i><code></i> in the input stream if the <i><relation></i> is true . After the <i><code></i> has been processed by T _E X the test will be repeated, and a loop will occur until the test is false .

6 Some useful constants, and scratch variables

<hr/> <code>\c_zero_fp</code> <code>\c_minus_zero_fp</code> <hr/>	Zero, with either sign.
New: 2012-05-08	
<hr/> <code>\c_one_fp</code> <hr/>	One as an fp: useful for comparisons in some places.
New: 2012-05-08	

<hr/> <code>\c_inf_fp</code> <code>\c_minus_inf_fp</code> <hr/>	Infinity, with either sign. These can be input directly in a floating point expression as <code>inf</code> and <code>-inf</code> .
<hr/> New: 2012-05-08 <hr/>	
<hr/> <code>\c_e_fp</code> <hr/>	The value of the base of the natural logarithm, $e = \exp(1)$.
<hr/> Updated: 2012-05-08 <hr/>	
<hr/> <code>\c_pi_fp</code> <hr/>	The value of π . This can be input directly in a floating point expression as <code>pi</code> . The value is rounded in a slightly odd way, to ensure for instance that <code>sin(pi)</code> yields an exact 0.
<hr/> Updated: 2012-05-08 <hr/>	
<hr/> <code>\c_one_degree_fp</code> <hr/>	The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians, suitable to be used for trigonometric functions. Within floating point expressions, this can be accessed as <code>deg</code> . Note that <code>180 deg = pi</code> exactly.
<hr/> New: 2012-05-08 <hr/>	
<hr/> <code>\l_tmpa_fp</code> <code>\l_tmpb_fp</code> <hr/>	Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.
<hr/> <code>\g_tmpa_fp</code> <code>\g_tmpb_fp</code> <hr/>	Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any L ^A T _E X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

7 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

“Exceptions” may occur when performing some floating point operations, such as `0 / 0`, or `10 ** 1e9999`. The IEEE standard defines 5 types of exceptions.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0 .
- *Invalid operation* occurs for operations with no defined outcome, for instance `0/0`, or `sin(∞)`, and almost any operation involving a `nan`. This normally results in a `nan`, except for conversion functions whose target type does not have a notion of `nan` (e.g., `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating e.g., `ln(0)` or `cot(0)`. This results in $\pm\infty$.

- *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 is associated a “flag”, which can be either *on* or *off*. By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions only raise the corresponding flag. The state of the flag can be tested and modified. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and turn the flag on, or only turn the flag on, or do nothing at all.

<code>\fp_if_flag_on_p:n</code> ★	<code>\fp_if_flag_on_p:n {<exception>}</code>
<code>\fp_if_flag_on:nTF</code> ★	<code>\fp_if_flag_on:nTF {<exception>} {<true code>} {<false code>}</code>
New: 2012-08-08	Tests if the flag for the <i><exception></i> is on, which normally means the given <i><exception></i> has occurred. <i>This function is experimental, and may be altered or removed.</i>
<code>\fp_flag_off:n</code>	<code>\fp_flag_off:n {<exception>}</code>
New: 2012-08-08	Locally turns off the flag which indicates whether the <i><exception></i> has occurred. <i>This function is experimental, and may be altered or removed.</i>
<code>\fp_flag_on:n</code> ★	<code>\fp_flag_on:n {<exception>}</code>
New: 2012-08-08	Locally turns on the flag to indicate (or pretend) that the <i><exception></i> has occurred. Note that this function is expandable: it is used internally by <code>l3fp</code> to signal when exceptions do occur. <i>This function is experimental, and may be altered or removed.</i>
<code>\fp_trap:nn</code>	<code>\fp_trap:nn {<exception>} {<trap type>}</code>
New: 2012-07-19 Updated: 2012-08-08	All occurrences of the <i><exception></i> (<i>invalid_operation</i> , <i>division_by_zero</i> , <i>overflow</i> , or <i>underflow</i>) within the current group are treated as <i><trap type></i> , which can be <ul style="list-style-type: none"> • none: the <i><exception></i> will be entirely ignored, and leave no trace; • flag: the <i><exception></i> will turn the corresponding flag on when it occurs; • error: additionally, the <i><exception></i> 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.

8 Viewing floating points

<code>\fp_show:N</code>	<code>\fp_show:N {<fp var>}</code>
<code>\fp_show:(c n)</code>	<code>\fp_show:n {<floating point expression>}</code>
New: 2012-05-08 Updated: 2012-08-14	Evaluates the <i><floating point expression></i> and displays the result in the terminal.

9 Floating point expressions

9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm 0.d_1d_2 \dots d_{16} \cdot 10^n$, a normal floating point number, with $d_i \in [0, 9]$, $d_1 \neq 0$, and $|n| \leq 10000$;
- ± 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);

(*not yet*) subnormal numbers $\pm 0.d_1d_2 \dots d_{16} \cdot 10^{-10000}$ with $d_1 = 0$.

Normal floating point numbers are stored in base 10, with 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 0 characters, and an optional . character), 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.

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 will be ignored.
- Any unrecognizable string triggers an error, and produces a **nan**.

Note that `e-1` is not a representation of 10^{-1} , because it could be mistaken with the difference of “`e`” and 1. This is consistent with several other programming languages. However, in order to avoid confusions, `e-1` is not considered to be this difference either. To input the base of natural logarithms, use `exp(1)` or `\c_e_fp`.

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.

- Implicit multiplication by juxtaposition (`2pi`, *etc*).
- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Binary `*`, `/` and `%`.
- 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

$$\begin{aligned}\text{sin2pi} &= \sin(2\pi) = 0, \\ 2^{\text{2max}(3,4)} &= 2^{2\max(3,4)} = 256.\end{aligned}$$

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 $\langle operand_2 \rangle$ if $\langle operand_1 \rangle$ is true, and $\langle operand_3 \rangle$ if it is false (equal to ± 0). All three $\langle operands \rangle$ 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.

```
TWOBARS \fp_eval:n { <operand1> || <operand2> }
```

If $\langle operand_1 \rangle$ is true (non-zero), use that value, otherwise the value of $\langle operand_2 \rangle$. Both $\langle operands \rangle$ 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> <comparison> <operand2> }
```

The $\langle comparison \rangle$ consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by `!`. It may not start with `?`. This evaluates to `+1` if the $\langle comparison \rangle$ between the $\langle operand_1 \rangle$ and $\langle operand_2 \rangle$ is true, and `+0` otherwise.

```
+ \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 `<operand>`, and `!` `<operand>` evaluates to 1 if `<operand>` 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 `<operand1>` to the power `<operand2>`. This operation is right associative, hence `2 ** 2 ** 3` equals `2^2^3 = 256`. The “invalid operation” exception occurs if `<operand1>` is negative or `-0`, and `<operand2>` is not an integer, unless the result is zero (in that case, the sign is chosen arbitrarily to be `+0`). “Division by zero” occurs when raising `±0` to a strictly negative power. “Underflow” and “overflow” occur when appropriate.

`abs` `\fp_eval:n { abs(<fpexpr>) }`

Computes the absolute value of the `<fpexpr>`. This function does not raise any exception beyond those raised when computing its operand `<fpexpr>`. See also `\fp_abs:n`.

`exp` `\fp_eval:n { exp(<fpexpr>) }`

Computes the exponential of the `<fpexpr>`. “Underflow” and “overflow” occur when appropriate.

`ln` `\fp_eval:n { ln(<fpexpr>) }`

Computes the natural logarithm of the `<fpexpr>`. 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) = -∞`. “Underflow” and “overflow” occur when appropriate.

`max` `\fp_eval:n { max(<fpexpr1> , <fpexpr2> , ...) }`
`min` `\fp_eval:n { min(<fpexpr1> , <fpexpr2> , ...) }`

Evaluates each `<fpexpr>` and computes the largest (smallest) of those. If any of the `<fpexpr>` is a `nan`, the result is `nan`. Those operations do not raise exceptions.

<code>round</code>	<code>\fp_eval:n { round <option> (<fpexpr>) }</code>
<code>round0</code>	<code>\fp_eval:n { round <option> (<fpexpr₁> , <fpexpr₂>) }</code>
<code>round+</code>	Rounds $\langle fpexpr_1 \rangle$ to $\langle fpexpr_2 \rangle$ places. When $\langle fpexpr_2 \rangle$ is omitted, it is assumed to be 0, <i>i.e.</i> , $\langle fpexpr_1 \rangle$ is rounded to an integer. The $\langle option \rangle$ controls the rounding direction:
<code>round-</code>	

- by default, the operation rounds to the closest allowed number (rounding ties to even);
- with 0, the operation rounds towards 0, *i.e.*, truncates;
- with +, the operation rounds towards $+\infty$;
- with -, the operation rounds towards $-\infty$.

If $\langle fpexpr_2 \rangle$ does not yield an integer less than 10^8 in absolute value, then an “invalid operation” exception is raised. “Overflow” may occur if the result is infinite (this cannot happen unless $\langle fpexpr_2 \rangle < -9984$).

<code>sin</code>	<code>\fp_eval:n { sin(<fpexpr>) }</code>
<code>cos</code>	<code>\fp_eval:n { cos(<fpexpr>) }</code>
<code>tan</code>	<code>\fp_eval:n { tan(<fpexpr>) }</code>
<code>cot</code>	<code>\fp_eval:n { cot(<fpexpr>) }</code>
<code>csc</code>	<code>\fp_eval:n { csc(<fpexpr>) }</code>
<code>sec</code>	<code>\fp_eval:n { sec(<fpexpr>) }</code>

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the $\langle fpexpr \rangle$. 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.

<code>inf</code>	The special values $+\infty$, $-\infty$, and <code>nan</code> are represented as <code>inf</code> , <code>-inf</code> and <code>nan</code> (see <code>\c_-nan</code> , <code>\c_minus_inf_fp</code> and <code>\c_nan_fp</code>).
<code>nan</code>	

<code>pi</code>	The value of π (see <code>\c_pi_fp</code>).
-----------------	--

<code>deg</code>	The value of 1° in radians (see <code>\c_one_degree_fp</code>).
------------------	---

—	
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	1cm = $\frac{1}{2.54}$ in = 28.45275590551181pt
mm	1mm = $\frac{1}{25.4}$ in = 2.845275590551181pt
dd	1dd = 0.376065mm = 1.07000856496063pt
cc	1cc = 12dd = 12.84010277952756pt
nd	1nd = 0.375mm = 1.066978346456693pt
nc	1nc = 12nd = 12.80374015748031pt
bp	1bp = $\frac{1}{72}$ in = 1.00375pt
sp	1sp = 2^{-16} pt = 1.52587890625e-5pt.
—	

The values of the (font-dependent) units `em` and `ex` are gathered from T_EX when the surrounding floating point expression is evaluated.

—	
true	Other names for 1 and +0.
false	
—	
<code>\dim_to_fp:n</code> ★	<code>\dim_to_fp:n {⟨dimexpr⟩}</code>
New: 2012-05-08	Expands to an internal floating point number equal to the value of the <i>⟨dimexpr⟩</i> in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, <code>\dim_to_fp:n</code> can be used to speed up parts of a computation where a low precision is acceptable.
—	
<code>\fp_abs:n</code> ★	<code>\fp_abs:n {⟨floating point expression⟩}</code>
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 {⟨fp expression 1⟩} {⟨fp expression 2⟩}</code>
<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 either a digit, or an underscore,
- the `\uccodes` are changed: the test for whether a character is a letter actually tests if the upper-case code of the character is between A and Z.

The following need to be done. I'll try to time-order the items.

- Decide what exponent range to consider.
- Change the internal representation of fp, by replacing braced groups of 4 digits by delimited arguments. Also consider changing the fp structure a bit to allow using `\pdfTEX_strcmp:D` to compare (not in LuaTeX: it is too slow)?
- Modulo and remainder, and rounding functions `quantize`, `quantize0`, `quantize+`, `quantize-`, `quantize=`, `round=`. Should the modulo also be provided as (catcode 12) `%`?
- `\fp_format:nn {<fpexpr>} {<format>}`, but what should `<format>` be? More general pretty printing?
- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?
- Add `csc` and `sec`.
- Add `log(x, b)` for logarithm of x in base b .
- `hypot` (Euclidean length) and `atan(x, y) = atan(x/y)`, also called `atan2` in other math packages. Cartesian-to-polar transform. Other inverse trigonometric functions `acos`, `asin`, `atan` (one and two arguments). Also `asec`, `acsc`?
- Hyperbolic functions `cosh`, `sinh`, `tanh`.
- Inverse hyperbolics.
- Base conversion, input such as `0xAB.CDEF`.
- Random numbers (pgfmath provides `rnd`, `rand`, `random`), with seed reset at every `\fp_set:Nn`.
- 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.
- Parse `-3 < -2 < -1` as it should, not `(-3 < -2) < -1`.
- 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?

`Pgfmath` also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs. (Exclamation points mark important bugs.)

- ! Some functions are not monotonic when they should. For instance, `sin(1 - 10-16)` is wrongly greater than `sin(1)`.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- `round` should accept any integer as its second argument.
- Logarithms of numbers very close to 1 are inaccurate.
- `tan` and `cot` give very slightly wrong results for arguments near 10^{-8} .
- When rounding towards $-\infty$, `\dim_to_fp:n {0pt}` should return -0 , not $+0$.
- The result of $(\pm 0) + (\pm 0)$ should depend on the rounding mode.
- `0e9999999999` gives a T_EX “number too large” error.
- Conversion to integers with `\fp_to_int:n` does not check for overflow.
- Subnormals are not implemented.
- `max(-inf)` will lose any information attached to this `-inf`.
- The overflow trap receives the wrong argument in `l3fp-expo` (see `exp(1e5678)` in `m3fp-traps001`).

Possible optimizations/improvements.

- Optimize argument reduction for trigonometric functions: we don’t need 6×4 digits here, only 4×4 .
- In subsection 9.1, write a grammar.
- Fix the TWO BARS business with the index.
- 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 , using methods similar to `__fp_fixed_div_to_float:ww`. However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (`l3fp-basics`).
- Understand and document `__fp_basics_pack_weird_low:NNNNw` and `__fp_basics_pack_weird_high:NNNNNNNNw` better. Move the other `basics_pack` auxiliaries to `l3fp-aux` under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan’s articles, some previous T_EX fp packages, the international standards,...
- Also take into account the “inexact” exception?

Part XXII

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 will raise an error: use `\luatex_if_engine:T` to avoid this. Details of coding the LuaTeX engine are detailed in the LuaTeX manual.

<code>\lua_now:n</code>	★	<code>\lua_now:n {⟨token list⟩}</code>
-------------------------	---	--

<code>\lua_now:x</code>	★
-------------------------	---

Updated: 2012-08-02

The `⟨token list⟩` is first tokenized by TeX, which will include 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 will execute the `⟨Lua input⟩` immediately, and in an expandable manner.

<code>\lua_now_x:n</code>	★	<code>\lua_now_x:n {⟨token list⟩}</code>
---------------------------	---	--

<code>\lua_now_x:x</code>	★
---------------------------	---

New: 2012-08-02

The `⟨token list⟩` is first tokenized and expanded by TeX, which will include 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_x:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the `⟨Lua input⟩` immediately, and in an expandable manner.

TeXhackers note: `\lua_now_x:n` is the LuaTeX primitive `\directlua` renamed.

<code>\lua_shipout:n</code>		<code>\lua_shipout:n {⟨token list⟩}</code>
-----------------------------	--	--

<code>\lua_shipout:x</code>	
-----------------------------	--

The `⟨token list⟩` is first tokenized by TeX, which will include 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.

TeXhackers note: At a TeX level, the `⟨Lua input⟩` is stored as a “whatsit”.

$\backslash\text{lua_shipout_x:n}$ $\backslash\text{lua_shipout_x:x}$	$\backslash\text{lua_shipout:n}$ $\{\langle\text{token list}\rangle\}$ <p>The $\langle\text{token list}\rangle$ is first tokenized by $\text{T}_{\text{E}}\text{X}$, which will include converting line ends to spaces in the usual $\text{T}_{\text{E}}\text{X}$ manner and which respects currently-applicable $\text{T}_{\text{E}}\text{X}$ category codes. The resulting $\langle\text{Lua input}\rangle$ is passed to the Lua interpreter when the current page is finalised (<i>i.e.</i> at shipout). Each $\backslash\text{lua_shipout:n}$ block is treated by Lua as a separate chunk. The Lua interpreter will execute the $\langle\text{Lua input}\rangle$ during the page-building routine: the $\langle\text{Lua input}\rangle$ is expanded during this process in addition to any expansion when the argument was read. This makes these functions suitable for including material finalised during the page building process (such as the page number).</p>
--	--

$\text{T}_{\text{E}}\text{X}$ hackers note: $\backslash\text{lua_shipout_x:n}$ is the $\text{LuaT}_{\text{E}}\text{X}$ primitive $\backslash\text{latelua}$ named using the $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}3$ scheme.

At a $\text{T}_{\text{E}}\text{X}$ level, the $\langle\text{Lua input}\rangle$ is stored as a “whatsit”.

2 Category code tables

As well as providing methods to break out into Lua, there are places where additional $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}3$ functions are provided by the $\text{LuaT}_{\text{E}}\text{X}$ engine. In particular, $\text{LuaT}_{\text{E}}\text{X}$ provides category code tables. These can be used to ensure that a set of category codes are in force in a more robust way than is possible with other engines. These are therefore used by $\backslash\text{ExplSyntaxOn}$ and ExplSyntaxOff when using the $\text{LuaT}_{\text{E}}\text{X}$ engine.

$\backslash\text{cctab_new:N}$	$\backslash\text{cctab_new:N}$ $\langle\text{category code table}\rangle$ <p>Creates a new category code table, initially with the codes as used by $\text{iniT}_{\text{E}}\text{X}$.</p>
---------------------------------	--

$\backslash\text{cctab_gset:Nn}$	$\backslash\text{cctab_gset:Nn}$ $\langle\text{category code table}\rangle$ $\{\langle\text{category code set up}\rangle\}$ <p>Sets the $\langle\text{category code table}\rangle$ to apply the category codes which apply when the prevailing régime is modified by the $\langle\text{category code set up}\rangle$. Thus within a standard code block the starting point will be the code applied by $\backslash\text{c_code_cctab}$. The assignment of the table is global: the underlying primitive does not respect grouping.</p>
-----------------------------------	---

$\backslash\text{cctab_begin:N}$	$\backslash\text{cctab_begin:N}$ $\langle\text{category code table}\rangle$ <p>Switches the category codes in force to those stored in the $\langle\text{category code table}\rangle$. The prevailing codes before the function is called are added to a stack, for use with $\backslash\text{cctab_end:}$.</p>
-----------------------------------	--

$\backslash\text{cctab_end:}$	$\backslash\text{cctab_end:}$ <p>Ends the scope of a $\langle\text{category code table}\rangle$ started using $\backslash\text{cctab_begin:N}$, retuning the codes to those in force before the matching $\backslash\text{cctab_begin:N}$ was used.</p>
--------------------------------	--

$\backslash\text{c_code_cctab}$	<p>Category code table for the code environment. This does not include setting the behaviour of the line-end character, which is only altered by $\backslash\text{ExplSyntaxOn}$.</p>
-----------------------------------	--

<u><code>\c_document_cctab</code></u>	Category code table for a standard L ^A T _E X document. This does not include setting the behaviour of the line-end character, which is only altered by <code>\ExplSyntaxOff</code> .
<u><code>\c_initex_cctab</code></u>	Category code table as set up by <code>iniT_EX</code> .
<u><code>\c_other_cctab</code></u>	Category code table where all characters have category code 12 (other).
<u><code>\c_str_cctab</code></u>	Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).

Part XXIII

The l3candidates package

Experimental additions to l3kernel

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.

1 Additions to l3basics

`\cs_if_exist_use:NTF` ★
`\cs_if_exist_use:cTF` ★

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

If the $\langle control\ sequence \rangle$ exists, leave it in the input stream, followed by the $\langle true\ code \rangle$ (unbraced). Otherwise, leave the $\langle false \rangle$ code in the input stream. For example,

```
\cs_set:Npn \mypkg_use_character:N #1
{ \cs_if_exist_use:cF { mypkg_#1:n } { \mypkg_default:N #1 } }
```

calls the function `\mypkg_#1:n` if it exists, and falls back to a default action otherwise. This could also be done (more slowly) using `\str_case_x:nnn`.

T_EXhackers note: The `c` variants do not introduce the $\langle control\ sequence \rangle$ in the hash table if it is not there.

2 Additions to l3box

2.1 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_resize:Nnn`
`\box_resize:cnn`

`\box_resize:Nnn` $\langle box \rangle$ $\{\langle x-size \rangle\}$ $\{\langle y-size \rangle\}$

Resize 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 vertical size (height plus depth) of the box. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. Negative sizes will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. The resizing applies within the current T_EX group level.

<code>\box_resize_to_ht_plus_dp:Nn</code>	<code>\box_resize_to_ht_plus_dp:Nn <box> {<y-size>}</code>
<code>\box_resize_to_ht_plus_dp:cn</code>	

Resize 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 vertical size (height plus depth) of the box. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative size will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. The resizing applies within the current \TeX group level.

<code>\box_resize_to_wd:Nn</code>	<code>\box_resize_to_wd:Nn <box> {<x-size>}</code>
<code>\box_resize_to_wd:cn</code>	

Resize 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 $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the resizing is applied. A negative size will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. The resizing applies within the current \TeX group level.

<code>\box_rotate:Nn</code>	<code>\box_rotate:Nn <box> {<angle>}</code>
<code>\box_rotate:cn</code>	

Rotates the $\langle box \rangle$ by $\langle angle \rangle$ (in degrees) anti-clockwise about its reference point. The reference point of the updated box will be moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the rotation is applied. The rotation applies within the current \TeX group level.

<code>\box_scale:Nnn</code>	<code>\box_scale:Nnn <box> {<x-scale>} {<y-scale>}</code>
<code>\box_scale:cnn</code>	

Scales the $\langle box \rangle$ by factors $\langle x-scale \rangle$ and $\langle y-scale \rangle$ in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the scaling is applied. Negative scalings will cause the material in the $\langle box \rangle$ to be reversed in direction, but the reference point of the $\langle box \rangle$ will be unchanged. The scaling applies within the current \TeX group level.

2.2 Viewing part of a box

<code>\box_clip:N</code>	<code>\box_clip:N <box></code>
<code>\box_clip:c</code>	

Clips the $\langle box \rangle$ in the output so that only material inside the bounding box is displayed in the output. The updated $\langle box \rangle$ will be an hbox, irrespective of the nature of the $\langle box \rangle$ before the clipping is applied. The clipping applies within the current \TeX group level.

These functions require the $\text{\LaTeX}3$ native drivers: they will not work with the $\text{\LaTeX}2_{\epsilon}$ graphics drivers!

\TeX hackers note: Clipping is implemented by the driver, and as such the full content of the box is places in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

<code>\box_trim:Nnnnn</code>	<code>\box_trim:Nnnnn <box> {\left} {\bottom} {\right} {\top}</code>
<code>\box_trim:cnnnn</code>	

Adjusts the bounding box of the `<box>` `<left>` is removed from the left-hand edge of the bounding box, `<right>` from the right-hand edge and so fourth. All adjustments are *<dimension expressions>*. Material output of the bounding box will still be displayed in the output unless `\box_clip:N` is subsequently applied. The updated `<box>` will be an hbox, irrespective of the nature of the `<box>` before the trim operation is applied. The adjustment applies within the current T_EX group level. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

<code>\box_viewport:Nnnnn</code>	<code>\box_viewport:Nnnnn <box> {\llx} {\lly} {\urx} {\ury}</code>
<code>\box_viewport:cnnnn</code>	

Adjusts the bounding box of the `<box>` such that it has lower-left co-ordinates (`<llx>`, `<lly>`) and upper-right co-ordinates (`<urx>`, `<ury>`). All four co-ordinate positions are *<dimension expressions>*. Material output of the bounding box will still be displayed in the output unless `\box_clip:N` is subsequently applied. The updated `<box>` will be an hbox, irrespective of the nature of the `<box>` before the viewport operation is applied. The adjustment applies within the current T_EX group level.

2.3 Internal variables

<code>\l__box_angle_fp</code>	The angle through which a box is rotated by <code>\box_rotate:Nn</code> , given in degrees counter-clockwise. This value is required by the underlying driver code in <code>l3driver</code> to carry out the driver-dependent part of box rotation.
-------------------------------	---

<code>\l__box_cos_fp</code>	The sine and cosine of the angle through which a box is rotated by <code>\box_rotate:Nn</code> : the values refer to the angle counter-clockwise. These values are required by the underlying driver code in <code>l3driver</code> to carry out the driver-dependent part of box rotation.
<code>\l__box_sin_fp</code>	

<code>\l__box_scale_x_fp</code>	The scaling factors by which a box is scaled by <code>\box_scale:Nnn</code> or <code>\box_resize:Nnn</code> . These values are required by the underlying driver code in <code>l3driver</code> to carry out the driver-dependent part of box rotation.
<code>\l__box_scale_y_fp</code>	

<code>\l__box_internal_box</code>	Box used for affine transformations, which is used to contain rotated material when applying <code>\box_rotate:Nn</code> . This box must be correctly constructed for the driver-dependent code in <code>l3driver</code> to function correctly.
-----------------------------------	---

3 Additions to l3clist

<hr/> <code>\clist_item:Nn</code> ★	<code>\clist_item:Nn <comma list> {<integer expression>}</code>
<code>\clist_item:(cn nn)</code> ★	Indexing items in the <i><comma list></i> from 1 at the top (left), this function will evaluate the <i><integer expression></i> and leave the appropriate item from the comma list in the input stream. If the <i><integer expression></i> is negative, indexing occurs from the bottom (right) of the comma list. When the <i><integer expression></i> is larger than the number of items in the <i><comma list></i> (as calculated by <code>\clist_count:N</code>) then the function will expand to nothing.

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

<hr/> <code>\clist_set_from_seq:NN</code>	<code>\clist_set_from_seq:NN <comma list> <sequence></code>
<code>\clist_set_from_seq:(cn Nc cc)</code>	
<hr/> <code>\clist_gset_from_seq:NN</code>	
<code>\clist_gset_from_seq:(cn Nc cc)</code>	

Sets the *<comma list>* to be equal to the content of the *<sequence>*. Items which contain either spaces or commas are surrounded by braces.

<hr/> <code>\clist_const:Nn</code>	<code>\clist_const:Nn <clist var> {<comma list>}</code>
<code>\clist_const:(Nx cn cx)</code>	Creates a new constant <i><clist var></i> or raises an error if the name is already taken. The value of the <i><clist var></i> will be set globally to the <i><comma list></i> .

<hr/> <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>

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.

4 Additions to l3coffins

<hr/> <code>\coffin_resize:Nnn</code>	<code>\coffin_resize:Nnn <coffin> {<width>} {<total-height>}</code>
<code>\coffin_resize:cnn</code>	Resized the <i><coffin></i> to <i><width></i> and <i><total-height></i> , both of which should be given as dimension expressions.
<hr/> <code>\coffin_rotate:Nn</code>	<code>\coffin_rotate:Nn <coffin> {<angle>}</code>
<code>\coffin_rotate:cn</code>	Rotates the <i><coffin></i> by the given <i><angle></i> (given in degrees counter-clockwise). This process will rotate both the coffin content and poles. Multiple rotations will 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.

5 Additions to l3file

`\ior_map_inline:Nn`

New: 2012-02-11

`\ior_map_inline:Nn` $\langle stream \rangle$ $\{\langle inline function \rangle\}$

Applies the $\langle inline function \rangle$ to $\langle lines \rangle$ obtained by reading one or more lines (until an equal number of left and right braces are found) from the $\langle stream \rangle$. The $\langle inline function \rangle$ should consist of code which will receive the $\langle line \rangle$ 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_str_map_inline:Nn`

New: 2012-02-11

`\ior_str_map_inline:Nn` $\{\langle stream \rangle\}$ $\{\langle inline function \rangle\}$

Applies the $\langle inline function \rangle$ to every $\langle line \rangle$ in the $\langle stream \rangle$. 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). The $\langle inline function \rangle$ should consist of code which will receive the $\langle line \rangle$ 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 $\langle stream \rangle$ have been processed. This will normally take 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 will lead 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 will depend 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 *<stream>* have been processed, inserting the *<tokens>* after the mapping has ended. This will normally take 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 will lead 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 the *<tokens>* are inserted into the input stream. This will depend on the design of the mapping function.

6 Additions to l3fp

\fp_set_from_dim:Nn**\fp_set_from_dim:cn****\fp_gset_from_dim:Nn****\fp_gset_from_dim:cn**

\fp_set_from_dim:Nn <floating point variable> {<dimexpr>}

Sets the *<floating point variable>* to the distance represented by the *<dimension expression>* in the units points. This means that distances given in other units are first converted to points before being assigned to the *<floating point variable>*.

7 Additions to l3prop

\prop_map_tokens:Nn ☆**\prop_map_tokens:cn** ☆

\prop_map_tokens:Nn <property list> {<code>}

Analogue of `\prop_map_function:NN` which maps several tokens instead of a single function. The *<code>* receives each key-value pair in the *<property list>* as two trailing brace groups. For instance,

```
\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }
```

will expand to the value corresponding to `mykey`: for each pair in `\l_my_prop` the function `\str_if_eq:nnT` receives `mykey`, the *<key>* and the *<value>* as its three arguments. For that specific task, `\prop_get:Nn` is faster.

<code>\prop_get:Nn</code> ★	<code>\prop_get:Nn</code> $\langle property\ list \rangle$ $\{\langle key \rangle\}$
<code>\prop_get:cn</code> ★	Expands to the $\langle value \rangle$ corresponding to the $\langle key \rangle$ in the $\langle property\ list \rangle$. If the $\langle key \rangle$ is missing, this has an empty expansion.

T_EXhackers 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 $\langle value \rangle$ will not expand further when appearing in an x-type argument expansion.

8 Additions to l3seq

<code>\seq_item:Nn</code> ★	<code>\seq_item:Nn</code> $\langle sequence \rangle$ $\{\langle integer\ expression \rangle\}$
<code>\seq_item:cn</code> ★	Indexing items in the $\langle sequence \rangle$ from 1 at the top (left), this function will evaluate the $\langle integer\ expression \rangle$ and leave 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. When the $\langle integer\ expression \rangle$ is larger than the number of items in the $\langle sequence \rangle$ (as calculated by <code>\seq_count:N</code>) then the function will expand to nothing.

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

<code>\seq_mapthread_function:NNN</code> ★	<code>\seq_mapthread_function:NNN</code> $\langle seq_1 \rangle$ $\langle seq_2 \rangle$ $\langle function \rangle$
<code>\seq_mapthread_function:(NcN cNN ccN)</code> ★	

Applies $\langle function \rangle$ to every pair of items $\langle seq_1\text{-}item \rangle$ – $\langle seq_2\text{-}item \rangle$ from the two sequences, returning items from both sequences from left to right. The $\langle function \rangle$ will receive two n-type arguments for each iteration. The mapping will terminate when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

<code>\seq_set_from_clist:NN</code>	<code>\seq_set_from_clist:NN</code> $\langle sequence \rangle$ $\langle comma\text{-}list \rangle$
<code>\seq_set_from_clist:(cN Nc cc Nn cn)</code>	
<code>\seq_gset_from_clist:NN</code>	
<code>\seq_gset_from_clist:(cN Nc cc Nn cn)</code>	

Sets the $\langle sequence \rangle$ within the current T_EX group to be equal to the content of the $\langle comma\text{-}list \rangle$.

<code>\seq_reverse:N</code>	<code>\seq_reverse:N</code> $\langle sequence \rangle$
<code>\seq_greverse:N</code>	Reverses the order of items in the $\langle sequence \rangle$, and assigns the result to $\langle sequence \rangle$, locally or globally according to the variant chosen.

`\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 $\langle inline\ boolexpr \rangle$ will receive 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 $\langle sequence_1 \rangle$.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level T_EX errors.

`\seq_set_map:NNn`
`\seq_gset_map:NNn`

New: 2011-12-22

`\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.

T_EXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level T_EX errors.

9 Additions to l3skip

`\dim_to_pt:n` ★

New: 2013-05-06

`\dim_to_pt:n` $\{\langle dimexpr \rangle\}$

Evaluates the $\langle dimension\ expression \rangle$, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by T_EX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

If the $\langle dimension\ expression \rangle$ contains additional tokens such as redundant units, these will be ignored, so for example

`\dim_to_pt:n { 1 bp pt }`

leaves 1.00374 in the input stream, *i.e.* the magnitude of one “big point” when converted to points.

`\dim_to_unit:nn` ★

New: 2013-05-06

`\dim_to_unit:nn` $\{\langle dimexpr_1 \rangle\}$ $\{\langle dimexpr_2 \rangle\}$

Evaluates the $\langle dimension\ expressions \rangle$, and leaves the value of $\langle dimexpr_1 \rangle$, expressed in a unit given by $\langle dimexpr_2 \rangle$, in the input stream. The result is a decimal number, rounded by T_EX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

If the $\langle dimension\ expressions \rangle$ contain additional tokens such as redundant units, these will be ignored, so for example

`\dim_to_unit:nn { 1 bp pt } { 1 mm }`

leaves 0.35277 in the input stream, *i.e.* the magnitude of one “big point” when converted to millimeters.

<code>\skip_split_finite_else_action:nnNN</code>	<code>\skip_split_finite_else_action:nnNN {<skipexpr>} {<action>}</code>
	<code><dimen₁> <dimen₂></code>

Checks if the `<skipexpr>` contains finite glue. If it does then it assigns `<dimen1>` the stretch component and `<dimen2>` the shrink component. If it contains infinite glue set `<dimen1>` and `<dimen2>` to 0pt and place #2 into the input stream: this is usually an error or warning message of some sort.

10 Additions to l3tl

<code>\tl_if_single_token_p:n</code> ★	<code>\tl_if_single_token_p:n {<token list>}</code>
<code>\tl_if_single_token:nTF</code> ★	<code>\tl_if_single_token:nTF {<token list>} {<true code>} {<false code>}</code>

Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single “normal” token. Token groups (`{...}`) are not single tokens.

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

This function, which works directly on T_EX tokens, reverses the order of the `<tokens>`: the first will be the last and the last will become first. Spaces are preserved. The reversal also operates within brace groups, but the braces themselves are not exchanged, as this would lead to an unbalanced token list. For instance, `\tl_reverse_tokens:n {a~{b()}}` leaves `{(b)~a` in the input stream. This function requires two steps of expansion.

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

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

Counts the number of T_EX tokens in the `<tokens>` and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of `a~{bc}` is 6. This function requires three expansions, giving an *<integer denotation>*.

<code>\tl_expandable_uppercase:n</code> ★	<code>\tl_expandable_uppercase:n {<tokens>}</code>
<code>\tl_expandable_lowercase:n</code> ★	<code>\tl_expandable_lowercase:n {<tokens>}</code>

The `\tl_expandable_uppercase:n` function works through all of the `<tokens>`, replacing characters in the range a–z (with arbitrary category code) by the corresponding letter in the range A–Z, with category code 11 (letter). Similarly, `\tl_expandable_lowercase:n` replaces characters in the range A–Z by letters in the range a–z, and leaves other tokens unchanged. This function requires two steps of expansion.

T_EXhackers note: Begin-group and end-group characters are normalized and become `{` and `}`, respectively. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the token list will not expand further when appearing in an x-type argument expansion.

<code>\tl_item:nn</code>	★	<code>\tl_item:nn {<token list>} {<integer expression>}</code>
<code>\tl_item:(Nn cn)</code>	★	

Indexing items in the *<token list>* from 1 on the left, this function will evaluate the *<integer expression>* and leave 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>* will not expand further when appearing in an *x*-type argument expansion.

11 Additions to l3tokens

<code>\char_set_active:Npn</code>	<code>\char_set_active:Npn <char> <parameters> {<code>}</code>
<code>\char_set_active:Npx</code>	

Makes *<char>* an active character to expand to *<code>* as replacement text. Within the *<code>*, the *<parameters>* (*#1, #2, etc.*) will be replaced by those absorbed. The *<char>* is made active within the current TeX group level, and the definition is also local.

<code>\char_gset_active:Npn</code>	<code>\char_gset_active:Npn <char> <parameters> {<code>}</code>
<code>\char_gset_active:Npx</code>	

Makes *<char>* an active character to expand to *<code>* as replacement text. Within the *<code>*, the *<parameters>* (*#1, #2, etc.*) will be replaced by those absorbed. The *<char>* is made active within the current TeX group level, but the definition is global. This function is therefore suited to cases where an active character definition should be applied only in some context (where the *<char>* is again made active).

<code>\char_set_active_eq:NN</code>	<code>\char_set_active_eq:NN <char> <function></code>
-------------------------------------	---

Makes *<char>* an active character equivalent in meaning to the *<function>* (which may itself be an active character). The *<char>* is made active within the current TeX group level, and the definition is also local.

<code>\char_gset_active_eq:NN</code>	<code>\char_gset_active_eq:NN <char> <function></code>
--------------------------------------	--

Makes *<char>* an active character equivalent in meaning to the *<function>* (which may itself be an active character). The *<char>* is made active within the current TeX group level, but the definition is global. This function is therefore suited to cases where an active character definition should be applied only in some context (where the *<char>* is again made active).

<hr/> <u><code>\peek_N_type:TF</code></u> <hr/>	<code>\peek_N_type:TF</code> $\{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
<hr/> <code>Updated: 2012-12-20</code> <hr/>	<p>Tests if the next $\langle token \rangle$ in the input stream can be safely grabbed as an N-type argument. The test will be $\langle false \rangle$ 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 <code>\c_space_token</code>, the test will take the $\langle false \rangle$ branch, even though the next $\langle token \rangle$ is in fact a valid N-type argument. The $\langle token \rangle$ will be left in the input stream after the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ (as appropriate to the result of the test).</p>

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
.....	<i>179</i>
**	<i>180</i>
?:	<i>179</i>
\:::	<i>34</i>
\::N	<i>34</i>
\::V	<i>34</i>
\::c	<i>34</i>
\::f	<i>34</i>
\::n	<i>34</i>
\::o	<i>34</i>
\::p	<i>34</i>
\::v	<i>34</i>
\::x	<i>34</i>
__chk_if_exist_cs:N	<i>24</i>
__chk_if_free_cs:N	<i>25</i>
__cs_count_signature:N	<i>25</i>
__cs_get_function_name:N	<i>25</i>
__cs_get_function_signature:N	<i>25</i>
__cs_split_function:NN	<i>25</i>
__cs_tmp:w	<i>25</i>
__dim_eval:w	<i>89</i>
__dim_eval_end:	<i>89</i>
__dim_strip_bp:n	<i>89</i>
__dim_strip_pt:n	<i>89</i>
__expl_package_check:	<i>7</i>
__file_name_sanitiz:n	<i>168</i>
__int_eval:w	<i>75</i>
__int_eval_end:	<i>75</i>
__int_to_roman:w	<i>74</i>
__int_value:w	<i>75</i>
__ior_open:Nn	<i>168</i>
__kernel_register_show:N	<i>25</i>
__msg_expandable_error:n	<i>149</i>
__msg_kernel_error:nnnnnn	<i>148</i>
__msg_kernel_expandable_error:nnnnnn	<i>148</i>
__msg_kernel_fatal:nnnnnn	<i>148</i>
__msg_kernel_info:nnnnnn	<i>148</i>
__msg_kernel_new:nnnn	<i>147</i>
__msg_kernel_set:nnnn	<i>147</i>
__msg_kernel_warning:nnnnnn	<i>148</i>
__msg_show_item:n	<i>149</i>
__msg_show_item_unbraced:nn	<i>149</i>
__msg_show_variable:Nnn	<i>149</i>
__msg_show_variable:n	<i>149</i>
__msg_term:nnnnnn	<i>149</i>
__my_map_dbl:nn	<i>4, 6, 11</i>
__my_map_dbl_fn:nn	<i>3, 10</i>
__prg_break:	<i>43</i>
__prg_break_point:	<i>43</i>
__prg_break_point:Nn	<i>43</i>
__prg_case_end:nw	<i>25</i>
__prg_compare_error:	<i>75</i>
__prg_map_break:Nn	<i>43</i>
__prg_variable_get_scope:N	<i>42</i>
__prg_variable_get_type:N	<i>42</i>
__prop_pair:wn	<i>127</i>
__prop_split:NnTF	<i>127</i>
__quark_if_recursion_tail_break:NN	<i>48</i>
__scan_new:N	<i>48</i>
__seq_item:n	<i>112</i>
__seq_pop_item_def:	<i>112</i>
__seq_push_item_def:n	<i>112</i>
__str_if_eq_x_return:nn	<i>26</i>
__tl_trim_spaces:nn	<i>103</i>
__use_none_delimit_by_s_stop:w	<i>48</i>
A	
false	<i>182</i>
nan	<i>181</i>
tan	<i>181</i>
max	<i>180</i>
B	
\bool_do_until:Nn	<i>40</i>
\bool_do_until:nn	<i>40</i>
\bool_do_while:Nn	<i>40</i>
\bool_do_while:nn	<i>40</i>
.bool_gset:c	<i>152</i>
.bool_gset:N	<i>152</i>
\bool_gset:Nn	<i>38</i>
\bool_gset_eq:NN	<i>38</i>
\bool_gset_false:N	<i>37</i>
.bool_gset_inverse:c	<i>152</i>
.bool_gset_inverse:N	<i>152</i>
\bool_gset_true:N	<i>38</i>
\bool_if:NnTF	<i>38</i>

\c_math_toggle_token	53	\char_set_catcode_active:N	50
\c_max_dim	82	\char_set_catcode_active:n	50
\c_max_int	73	\char_set_catcode_alignment:N	50
\c_max_muskip	88	\char_set_catcode_alignment:n	50
\c_max_register_int	73	\char_set_catcode_comment:N	50
\c_max_skip	85	\char_set_catcode_comment:n	50
\c_minus_inf_fp	175	\char_set_catcode_end_line:N	50
\c_minus_one	73	\char_set_catcode_end_line:n	50
\c_minus_zero_fp	174	\char_set_catcode_escape:N	50
\c_nine	73	\char_set_catcode_escape:n	50
\c_one	73	\char_set_catcode_group_begin:N	50
\c_one_degree_fp	175	\char_set_catcode_group_begin:n	50
\c_one_fp	174	\char_set_catcode_group_end:N	50
\c_one_hundred	73	\char_set_catcode_group_end:n	50
\c_one_thousand	73	\char_set_catcode_ignore:N	50
\c_other_cctab	188	\char_set_catcode_ignore:n	50
\c_parameter_token	53	\char_set_catcode_invalid:N	50
\c_pi_fp	175	\char_set_catcode_invalid:n	50
\c_seven	73	\char_set_catcode_letter:N	50
\c_six	73	\char_set_catcode_letter:n	50
\c_sixteen	73	\char_set_catcode_math_subscript:N	50
\c_space_tl	103	\char_set_catcode_math_subscript:n	50
\c_space_token	53	\char_set_catcode_math_superscript:N	50
\c_str_cctab	188	\char_set_catcode_math_superscript:n	50
\c_ten	73	\char_set_catcode_math_toggle:N	50
\c_ten_thousand	73	\char_set_catcode_math_toggle:n	50
\c_term_ior	167	\char_set_catcode_other:N	50
\c_term_iow	167	\char_set_catcode_other:n	50
\c_thirteen	73	\char_set_catcode_parameter:N	50
\c_thirty_two	73	\char_set_catcode_parameter:n	50
\c_three	73	\char_set_catcode_space:N	50
\c_true_bool	21	\char_set_catcode_space:n	50
\c_twelve	73	\char_set_lccode:nn	51
\c_two	73	\char_set_mathcode:nn	52
\c_two_hundred_fifty_five	73	\char_set_sfcode:nn	52
\c_two_hundred_fifty_six	73	\char_set_uccode:nn	52
\c_zero	73	\char_show_value_catcode:n	51
\c_zero_dim	82	\char_show_value_lccode:n	51
\c_zero_fp	174	\char_show_value_mathcode:n	52
\c_zero_muskip	88	\char_show_value_sfcode:n	53
\c_zero_skip	85	\char_show_value_uccode:n	52
\cctab_begin:N	187	\char_value_catcode:n	51
\cctab_end:	187	\char_value_lccode:n	51
\cctab_gset:Nn	187	\char_value_mathcode:n	52
\cctab_new:N	187	\char_value_sfcode:n	52
\char_gset_active:Npn	198	\char_value_uccode:n	52
\char_gset_active_eq:NN	198	.choice:	152
\char_set_active:Npn	198	.choices:nn	152
\char_set_active_eq:NN	198	.choices:on	152
\char_set_catcode:nn	51	.choices:Vn	152

.choices:xn	152	\clist_show:n	121
\clist_clear:N	113	\clist_use:Nn	119
\clist_clear_new:N	113	\clist_use:Nnnn	119
\clist_concat:NNN	114	.code:n	152
\clist_const:Nn	192	\coffin_attach:NnnNnnnn	138
\clist_count:N	118	\coffin_clear:N	136
\clist_gclear:N	113	\coffin_display_handles:Nn	139
\clist_gclear_new:N	113	\coffin_dp:N	138
\clist_gconcat:NNN	114	\coffin_ht:N	139
\clist_get:NN	120	\coffin_if_exist:NTF	136
\clist_get:NNTF	120	\coffin_if_exist_p:N	136
\clist_gpop:NN	120	\coffin_join:NnnNnnnn	138
\clist_gpop:NNTF	120	\coffin_mark_handle:Nnnn	139
\clist_gpush:Nn	121	\coffin_new:N	136
\clist_gput_left:Nn	114	\coffin_resize:Nnn	192
\clist_gput_right:Nn	114	\coffin_rotate:Nn	192
\clist_gremove_all:Nn	115	\coffin_scale:Nnn	193
\clist_gremove_duplicates:N	115	\coffin_set_eq:NN	136
.clist_gset:c	152	\coffin_set_horizontal_pole:Nnn	137
.clist_gset:N	152	\coffin_set_vertical_pole:Nnn	137
\clist_gset:Nn	114	\coffin_show_structure:N	139
\clist_gset_eq:NN	114	\coffin_typeset:Nnnnn	138
\clist_gset_from_seq:NN	192	\coffin_wd:N	139
\clist_if_empty:NTF	115	\color_ensure_current:	140
\clist_if_empty:nTF	192	\color_group_begin:	140
\clist_if_empty_p:N	115	\color_group_end:	140
\clist_if_empty_p:n	192	\cs:w	17
\clist_if_exist:NTF	114	\cs_end:	17
\clist_if_exist_p:N	114	\cs_generate_from_arg_count:NNnn	15
\clist_if_in:NnTF	116	\cs_generate_variant:Nn	28
\clist_item:Nn	192	\cs_gset:Nn	14
\clist_map_break:	117	\cs_gset:Npn	12
\clist_map_break:n	118	\cs_gset_eq:NN	16
\clist_map_function:NN	116	\cs_gset_nopar:Nn	14
\clist_map_inline:Nn	116	\cs_gset_nopar:Npn	12
\clist_map_variable:NNn	117	\cs_gset_protected:Nn	15
\clist_new:N	113	\cs_gset_protected:Npn	12
\clist_pop:NN	120	\cs_gset_protected_nopar:Nn	15
\clist_pop:NNTF	120	\cs_gset_protected_nopar:Npn	13
\clist_push:Nn	121	\cs_if_eq:NNTF	21
\clist_put_left:Nn	114	\cs_if_eq_p:NN	21
\clist_put_right:Nn	114	\cs_if_exist:NTF	21
\clist_remove_all:Nn	115	\cs_if_exist_p:N	21
\clist_remove_duplicates:N	115	\cs_if_exist_use:NTF	17, 17, 189
.clist_set:c	152	\cs_if_free:NTF	22
.clist_set:N	152	\cs_if_free_p:N	22
\clist_set:Nn	114	\cs_meaning:N	16
\clist_set_eq:NN	114	\cs_new:Nn	6, 13
\clist_set_from_seq:NN	192	\cs_new:Npn	1, 11
\clist_show:N	121	\cs_new_eq:NN	15

\cs_new_nopar:Nn	13	\dim_min:nn	77
\cs_new_nopar:Npn	11	\dim_new:N	76
\cs_new_protected:Nn	13	\dim_ratio:nn	78
\cs_new_protected:Npn	11	.dim_set:c	153
\cs_new_protected_nopar:Nn	13	.dim_set:N	153
\cs_new_protected_nopar:Npn	11	\dim_set:Nn	77
\cs_set:Nn	14	\dim_set_eq:NN	77
\cs_set:Npn	3, 11	\dim_show:N	82
\cs_set_eq:NN	16	\dim_show:n	82
\cs_set_nopar:Nn	14	\dim_sub:Nn	77
\cs_set_nopar:Npn	12	\dim_to_fp:n	182
\cs_set_protected:Nn	14	\dim_to_pt:n	196
\cs_set_protected:Npn	12	\dim_to_unit:nn	196
\cs_set_protected_nopar:Nn	14	\dim_until_do:nn	81
\cs_set_protected_nopar:Npn	12	\dim_until_do:nNnn	81
\cs_show:N	16	\dim_use:N	82
\cs_to_str:N	4, 18	\dim_while_do:nn	81
\cs_undefine:N	16	\dim_while_do:nNnn	81
		\dim_zero:N	76
		\dim_zero_new:N	76
D			
dd	182		
nd	182		
.default:n	153		
.default:o	153		
.default:V	153		
.default:x	153		
\dim_abs:n	77		
\dim_add:Nn	77		
\dim_case:nnTF	80		
\dim_compare:nNnTF	78		
\dim_compare:nTF	79		
\dim_compare_p:n	79		
\dim_compare_p:nNn	78		
\dim_const:Nn	76		
\dim_do_until:nn	81		
\dim_do_until:nNnn	80		
\dim_do_while:nn	81		
\dim_do_while:nNnn	80		
\dim_eval:n	81		
\dim_gadd:Nn	77		
.dim_gset:c	153		
.dim_gset:N	153		
\dim_gset:Nn	77		
\dim_gset_eq:NN	77		
\dim_gsub:Nn	77		
\dim_gzero:N	76		
\dim_gzero_new:N	76		
\dim_if_exist:NnTF	76		
\dim_if_exist_p:N	76		
\dim_max:nn	77		
		E	
		sec	181
		deg	181
		\else:	24
		\exp_after:wN	32
		\exp_args:Nc	29
		\exp_args:Nf	30
		\exp_args:NNNo	31
		\exp_args:NNnx	31
		\exp_args:NNo	30
		\exp_args:Nno	30
		\exp_args:NNoo	31
		\exp_args:NNx	31
		\exp_args:No	29
		\exp_args:NV	30
		\exp_args:Nv	30
		\exp_args:Nx	30
		\exp_last_two_unbraced:Noo	32
		\exp_last_unbraced:Nf	32
		\exp_last_unbraced:Nx	32
		\exp_not:c	33
		\exp_not:f	33
		\exp_not:N	33
		\exp_not:n	33
		\exp_not:o	33
		\exp_not:V	33
		\exp_not:v	33
		\exp_stop_f:	33
		\ExplSyntaxOff	4, 6

\hbox_overlap_right:n	132	\int_decr:N	64
\hbox_set:Nn	132	\int_div_round:nn	62
\hbox_set:Nw	133	\int_div_truncate:nn	63
\hbox_set_end:	133	\int_do_until:nn	68
\hbox_set_to_wd:Nnn	132	\int_do_until:nNnn	67
\hbox_to_wd:nn	132	\int_do_while:nn	68
\hbox_to_zero:n	132	\int_do_while:nNnn	68
\hbox_unpack:N	133	\int_eval:n	62
\hbox_unpack_clear:N	133	\int_from_alph:n	71
\hcoffin_set:Nn	136	\int_from_base:nn	72
\hcoffin_set:Nw	137	\int_from_binary:n	71
\hcoffin_set_end:	137	\int_from_hexadecimal:n	71
		\int_from_octal:n	72
		\int_from_roman:n	72
		\int_gadd:Nn	64
pi	181	\int_gdecr:N	64
\if:w	24	\int_gincr:N	64
\if_bool:N	42	\int_gset:c	153
\if_box_empty:N	135	\int_gset:N	153
\if_case:w	74	\int_gset:Nn	64
\if_catcode:w	24	\int_gset_eq:NN	63
\if_charcode:w	24	\int_gsub:Nn	64
\if_cs_exist:N	24	\int_gzero:N	63
\if_dim:w	88	\int_gzero_new:N	63
\if_eof:w	167	\int_if_even:nTF	67
\if_false:	24	\int_if_even_p:n	67
\if_hbox:N	135	\int_if_exist:nTF	64
\if_int_compare:w	74	\int_if_exist_p:N	64
\if_int_odd:w	74	\int_if_odd:nTF	67
\if_meaning:w	24	\int_if_odd_p:n	67
\if_mode_horizontal:	24	\int_incr:N	64
\if_mode_inner:	24	\int_max:nn	63
\if_mode_math:	24	\int_min:nn	63
\if_mode_vertical:	24	\int_mod:nn	63
\if_predicate:w	42	\int_new:N	63
\if_true:	24	\int_set:c	153
\if_vbox:N	135	\int_set:N	153
min	180	\int_set:Nn	64
sin	181	\int_set_eq:NN	63
.initial:n	153	\int_show:N	72
.initial:o	153	\int_show:n	72
.initial:V	153	\int_step_function:nnnN	69
.initial:x	153	\int_step_inline:nnnn	69
\int_abs:n	62	\int_step_variable:nnnNn	69
\int_add:Nn	64	\int_sub:Nn	64
\int_case:nnTF	67	\int_to_Alph:n	70
\int_compare:nNnTF	65	\int_to_alph:n	70
\int_compare:nTF	66	\int_to_arabic:n	69
\int_compare_p:n	66	\int_to_base:nn	71
\int_compare_p:nNn	65	\int_to_binary:n	70
\int_const:Nn	63		

<code>\lua_shipout_x:n</code>	187	<code>\muskip_const:Nn</code>	86
<code>\luatex_if_engine:TF</code>	23	<code>\muskip_eval:n</code>	87
<code>\luatex_if_engine_p:</code>	23	<code>\muskip_gadd:Nn</code>	87
M			
<code>cm</code>	182	<code>\muskip_gset:Nn</code>	87
<code>em</code>	182	<code>\muskip_gset_eq:NN</code>	87
<code>mm</code>	182	<code>\muskip_gsub:Nn</code>	87
<code>.meta:n</code>	154	<code>\muskip_gzero:N</code>	86
<code>.meta:nn</code>	154	<code>\muskip_gzero_new:N</code>	86
<code>\mode_if_horizontal:TF</code>	41	<code>\muskip_if_exist:NTF</code>	86
<code>\mode_if_horizontal_p:</code>	41	<code>\muskip_if_exist_p:N</code>	86
<code>\mode_if_inner:TF</code>	41	<code>\muskip_new:N</code>	86
<code>\mode_if_inner_p:</code>	41	<code>\muskip_set:Nn</code>	87
<code>\mode_if_math:TF</code>	41	<code>\muskip_set_eq:NN</code>	87
<code>\mode_if_math_p:</code>	41	<code>\muskip_show:N</code>	88
<code>\mode_if_vertical:TF</code>	41	<code>\muskip_show:n</code>	88
<code>\mode_if_vertical_p:</code>	41	<code>\muskip_sub:Nn</code>	87
<code>\msg_critical:nnnnnn</code>	144	<code>\muskip_use:N</code>	87
<code>\msg_critical_text:n</code>	142	<code>\muskip_zero:N</code>	86
<code>\msg_error:nnnnnn</code>	144	<code>\muskip_zero_new:N</code>	86
<code>\msg_error_text:n</code>	142	<code>\my_map_dbl:nn</code>	1
<code>\msg_fatal:nnnnnn</code>	143	N	
<code>\msg_fatal_text:n</code>	142	<code>in</code>	182
<code>\msg_gset:nnnn</code>	142	<code>ln</code>	180
<code>\msg_if_exist:nnTF</code>	142	<code>inf</code>	181
<code>\msg_if_exist_p:nn</code>	142	O	
<code>\msg_info:nnnnnn</code>	144	<code>\or:</code>	24, 74
<code>\msg_info_text:n</code>	143	<code>cos</code>	181
<code>\msg_interrupt:nnn</code>	146	<code>cot</code>	181
<code>\msg_line_context:</code>	142	<code>round</code>	181
<code>\msg_line_number:</code>	142	<code>round+</code>	181
<code>\msg_log:n</code>	147	<code>round-</code>	181
<code>\msg_log:nnnnnn</code>	144	<code>round0</code>	181
<code>\msg_new:nnnn</code>	141	P	
<code>\msg_none:nnnnnn</code>	145	<code>bp</code>	182
<code>\msg_redirect_class:nn</code>	145	<code>sp</code>	182
<code>\msg_redirect_module:nnn</code>	146	<code>\pdfTeX_if_engine:TF</code>	23
<code>\msg_redirect_name:nnn</code>	146	<code>\pdfTeX_if_engine_p:</code>	23
<code>\msg_see_documentation_text:n</code>	143	<code>\peek_after:Nw</code>	58
<code>\msg_set:nnnn</code>	142	<code>\peek_catcode:NTF</code>	58
<code>\msg_term:n</code>	147	<code>\peek_catcode_ignore_spaces:NTF</code>	58
<code>\msg_warning:nnnnnn</code>	144	<code>\peek_catcode_remove:NTF</code>	59
<code>\msg_warning_text:n</code>	143	<code>\peek_catcode_remove_ignore_spaces:NTF</code>	59
<code>.multichoice:</code>	154	<code>\peek_charcode:NTF</code>	59
<code>.multichoices:nn</code>	154	<code>\peek_charcode_ignore_spaces:NTF</code>	59
<code>.multichoices:on</code>	154	<code>\peek_charcode_remove:NTF</code>	59
<code>.multichoices:Vn</code>	154		
<code>.multichoices:xn</code>	154		
<code>\muskip_add:Nn</code>	87		

\tl_expandable_uppercase:n	197	\tl_if_single_token:nTF	197
\tl_gclear:N	91	\tl_if_single_token_p:n	197
\tl_gclear_new:N	91	\tl_item:nn	198
\tl_gconcat:NNN	91	\tl_map_break:	97
\tl_gput_left:Nn	92	\tl_map_break:n	97
\tl_gput_right:Nn	92	\tl_map_function:NN	96
\tl_gremove_all:Nn	93	\tl_map_function:nN	96
\tl_gremove_once:Nn	92	\tl_map_inline:Nn	96
\tl_greplace_all:Nnn	92	\tl_map_inline:nn	96
\tl_greplace_once:Nnn	92	\tl_map_variable:NNn	96
\tl_greverse:N	99	\tl_map_variable:nNn	96
.tl_gset:c	154	\tl_new:N	91
.tl_gset:N	154	\tl_put_left:Nn	92
\tl_gset:Nn	92	\tl_put_right:Nn	92
\tl_gset_eq:NN	91	\tl_remove_all:Nn	93
\tl_gset_rescan:Nnn	93	\tl_remove_once:Nn	92
.tl_gset_x:c	154	\tl_replace_all:Nnn	92
.tl_gset_x:N	154	\tl_replace_once:Nnn	92
\tl_gtrim_spaces:N	99	\tl_rescan:nn	93
\tl_head:N	100	\tl_reverse:N	99
\tl_head:w	100	\tl_reverse:n	98
\tl_if_blank:nTF	94	\tl_reverse_items:n	99
\tl_if_blank_p:n	94	\tl_reverse_tokens:n	197
\tl_if_empty:nTF	94	.tl_set:c	154
\tl_if_empty:nTF	94	.tl_set:N	154
\tl_if_empty_p:N	94	\tl_set:Nn	92
\tl_if_empty_p:n	94	\tl_set_eq:NN	91
\tl_if_eq:NNTF	94	\tl_set_rescan:Nnn	93
\tl_if_eq:nnTF	94	.tl_set_x:c	154
\tl_if_eq_p:NN	94	.tl_set_x:N	154
\tl_if_exist:nTF	91	\tl_show:N	102
\tl_if_exist_p:N	91	\tl_show:n	102
\tl_if_head_eq_catcode:nNTF	101	\tl_tail:N	101
\tl_if_head_eq_catcode_p:nN	101	\tl_to_lowercase:n	93
\tl_if_head_eq_charcode:nNTF	101	\tl_to_str:N	97
\tl_if_head_eq_charcode_p:nN	101	\tl_to_str:n	98
\tl_if_head_eq_meaning:nNTF	101	\tl_to_uppercase:n	94
\tl_if_head_eq_meaning_p:nN	101	\tl_trim_spaces:N	99
\tl_if_head_is_group:nTF	102	\tl_trim_spaces:n	99
\tl_if_head_is_group_p:n	102	\tl_use:N	98
\tl_if_head_is_N_type:nTF	102	\token_get_arg_spec:N	61
\tl_if_head_is_N_type_p:n	102	\token_get_prefix_spec:N	61
\tl_if_head_is_space:nTF	102	\token_get_replacement_spec:N	61
\tl_if_head_is_space_p:n	102	\token_if_active:nTF	55
\tl_if_in:NnTF	95	\token_if_active_p:N	55
\tl_if_in:nnTF	95	\token_if_alignment:nTF	54
\tl_if_single:nTF	95	\token_if_alignment_p:N	54
\tl_if_single:nTF	95	\token_if_chardef:nTF	56
\tl_if_single_p:N	95	\token_if_chardef_p:N	56
\tl_if_single_p:n	95	\token_if_cs:nTF	56

	W		<code>\xetex_if_engine:TF</code>	5, 23
TWOBARS		179	<code>\xetex_if_engine_p:</code>	23
	X		<code>exp</code>	180
ex		182		