

The L^AT_EX3 Sources

The L^AT_EX3 Project*

April 27, 2011

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.

*Frank Mittelbach, Denys Duchier, Chris Rowley, Rainer Schöpf, Johannes Braams, Michael Downes, David Carlisle, Alan Jeffrey, Morten Høgholm, Thomas Lotze, Javier Bezos, Will Robertson, Joseph Wright

Contents

I	Introduction to <code>expl3</code> and this document	1
1	Naming functions and variables	1
1.0.1	Terminological inexactitude	3
2	Documentation conventions	3
II	The <code>l3names</code> package: A systematic naming scheme for <code>TeX</code>	5
3	Setting up the <code>LaTeX3</code> programming language	5
4	Using the modules	5
III	The <code>l3basics</code> package: Basic Definitions	7
5	Predicates and conditionals	7
5.1	Primitive conditionals	8
5.2	Non-primitive conditionals	10
6	Control sequences	12
7	Selecting and discarding tokens from the input stream	12
7.1	Extending the interface	14
7.2	Selecting tokens from delimited arguments	14
8	That which belongs in other modules but needs to be defined earlier	15
9	Defining functions	16
9.1	Defining new functions using primitive parameter text	17
9.2	Defining new functions using the signature	18
9.3	Defining functions using primitive parameter text	19
9.4	Defining functions using the signature (no checks)	21

9.5	Undefining functions	22
9.6	Copying function definitions	22
9.7	Internal functions	23
10	The innards of a function	24
11	Grouping and scanning	25
12	Checking the engine	25
IV	The <code>l3expan</code> package: Controlling Expansion of Function Arguments	25
13	Brief overview	26
14	Defining new variants	26
14.1	Methods for defining variants	27
15	Introducing the variants	28
16	Manipulating the first argument	29
17	Manipulating two arguments	30
18	Manipulating three arguments	31
19	Preventing expansion	31
20	Unbraced expansion	32
V	The <code>l3prg</code> package: Program control structures	33
21	Conditionals and logical operations	33
22	Defining a set of conditional functions	33
23	The boolean data type	35

24 Boolean expressions	36
25 Case switches	38
26 Generic loops	39
27 Choosing modes	39
28 Alignment safe grouping and scanning	40
29 Producing n copies	40
30 Sorting	41
30.1 Variable type and scope	42
30.2 Mapping to variables	42
VI The l3quark package: “Quarks”	43
31 Functions	43
32 Recursion	44
33 Constants	45
VII The l3token package: A token of my appreciation...	45
34 Character tokens	46
35 Generic tokens	49
35.1 Useless code: because we can!	53
36 Peeking ahead at the next token	53
VIII The l3int package: Integers/counters	55

37 Integer values	55
37.1 Integer expressions	55
37.2 Integer variables	56
37.3 Comparing integer expressions	59
37.4 Formatting integers	61
37.5 Converting from other formats	62
37.6 Low-level conversion functions	63
38 Variables and constants	65
38.1 Internal functions	66
 IX The l3skip package: Dimension and skip registers	 68
39 Skip registers	68
39.1 Functions	68
39.2 Formatting a skip register value	71
39.3 Variable and constants	71
40 Dim registers	71
40.1 Functions	71
40.2 Variable and constants	75
41 Muskip	75
 X The l3tl package: Token Lists	 76
42 Functions	76
43 Predicates and conditionals	80
44 Working with the contents of token lists	81
45 Variables and constants	83
46 Searching for and replacing tokens	84

47 Heads or tails?	85
XI The l3toks package: Token Registers	86
48 Allocation and use	86
49 Adding to the contents of token registers	89
50 Predicates and conditionals	90
51 Variable and constants	90
XII	90
52 Creating and initialising sequences	91
53 Appending data to sequences	92
54 Recovering items from sequences	93
55 Modifying sequences	95
55.1 Sequence conditionals	96
56 Mapping to sequences	96
56.1 Sequences as stacks	98
57 Viewing sequences	99
XIII The l3clist package: Comma separated lists	99
58 Functions for creating/initialising comma-lists	100
59 Putting data in	101
60 Getting data out	102
61 Mapping functions	103

62 Predicates and conditionals	104
63 Higher level functions	105
64 Functions for ‘comma-list stacks’	106
65 Internal functions	107
 XIV The <code>l3prop</code> package: Property Lists	 107
66 Functions	108
67 Predicates and conditionals	110
68 Internal functions	111
 XV The <code>l3box</code> package: Boxes	 112
69 Generic functions	112
70 Horizontal mode	116
71 Vertical mode	117
 XVI The <code>l3io</code> package: Low-level file i/o	 118
72 Opening and closing streams	119
72.1 Writing to files	120
72.2 Reading from files	122
73 Internal functions	122
74 Variables and constants	122
 XVII The <code>l3msg</code> package: Communicating with the user	 123

75 Creating new messages	124
76 Message classes	124
77 Redirecting messages	126
78 Support functions for output	127
79 Low-level functions	127
80 Kernel-specific functions	128
81 Variables and constants	130
 XVIII The l3xref package: Cross references	 130
 XIX The l3keyval package: Key-value parsing	 131
82 Features of l3keyval	132
83 Functions for keyval processing	132
84 Internal functions	133
85 Variables and constants	134
 XX The l3keys package: Key-value support	 134
86 Creating keys	136
87 Sub-dividing keys	139
87.1 Multiple choices	140
88 Setting keys	141
88.1 Examining keys: internal representation	142
89 Internal functions	142

90 Variables and constants	144
XXI	145
XXII The <code>l3fp</code> package: Floating point arithmetic	146
91 Floating point numbers	147
91.1 Constants	147
91.2 Floating-point variables	148
91.3 Conversion to other formats	149
91.4 Rounding floating point values	150
91.5 Tests on floating-point values	151
91.6 Unary operations	152
91.7 Arithmetic operations	152
91.8 Power operations	154
91.9 Exponential and logarithm functions	154
91.10 Trigonometric functions	155
91.11 Notes on the floating point unit	155
XXIII The <code>l3luatex</code> package: Lua_{TeX}-specific functions	156
92 Breaking out to Lua	156
93 Category code tables	157
XXIV Implementation	158
94 <code>l3names</code> implementation	158
94.1 Internal functions	159
94.2 Bootstrap code	159
94.3 Requirements	160
94.4 Catcode assignments	161

94.5	Setting up primitive names	162
94.6	Reassignment of primitives	163
94.7	expl3 code switches	174
94.8	Package loading	175
94.9	Finishing up	179
94.10	Showing memory usage	181
95	l3basics implementation	182
95.1	Renaming some T _E X primitives (again)	182
95.2	Defining functions	184
95.3	Selecting tokens	185
95.4	Gobbling tokens from input	187
95.5	Expansion control from l3expan	187
95.6	Conditional processing and definitions	187
95.7	Dissecting a control sequence	192
95.8	Exist or free	195
95.9	Defining and checking (new) functions	197
95.10	More new definitions	200
95.11	Copying definitions	202
95.12	Undefining functions	203
95.13	Diagnostic wrapper functions	204
95.14	Engine specific definitions	204
95.15	Scratch functions	204
95.16	Defining functions from a given number of arguments	205
95.17	Using the signature to define functions	207
96	l3expan implementation	209
96.1	Internal functions and variables	209
96.2	Module code	210
96.3	General expansion	211
96.4	Hand-tuned definitions	215
96.5	Definitions with the ‘general’ technique	216

96.6 Preventing expansion	217
96.7 Defining function variants	217
96.8 Last-unbraced versions	220
96.9 Items held from earlier	221
97 l3prg implementation	222
97.1 Variables	222
97.2 Module code	222
97.3 Choosing modes	223
97.4 Producing n copies	224
97.5 Booleans	228
97.6 Parsing boolean expressions	229
97.7 Case switch	236
97.8 Sorting	238
97.9 Variable type and scope	241
97.10 Mapping to variables	241
98 l3quark implementation	243
99 l3token implementation	247
99.1 Documentation of internal functions	247
99.2 Module code	247
99.3 Character tokens	248
99.4 Generic tokens	250
99.5 Peeking ahead at the next token	259
100 l3int implementation	266
100.1 Internal functions and variables	266
100.2 Module loading and primitives definitions	266
100.3 Allocation and setting	267
100.4 Scanning and conversion	275
100.5 Defining constants	286
100.6 Backwards compatibility	288

101	skip implementation	289
101.1	Skip registers	289
101.2	Dimen registers	293
101.3	Muskips	298
102	tl implementation	299
102.1	Functions	299
102.2	Variables and constants	304
102.3	Predicates and conditionals	306
102.4	Working with the contents of token lists	308
102.5	Checking for and replacing tokens	314
102.6	Heads or tails?	316
103	toks implementation	318
103.1	Allocation and use	318
103.2	Adding to token registers' contents	321
103.3	Predicates and conditionals	322
103.4	Variables and constants	323
104	Internal sequence functions	324
105	Sequence implementation	325
105.1	Allocation and initialisation	325
105.2	Appending data to either end	326
105.3	Breaking sequence functions	327
105.4	Mapping to sequences	328
105.5	Sequence stacks	329
105.6	Sequence conditionals	333
105.7	Modifying sequences	334
106	Viewing sequences	335
106.1	Depreciated interfaces	336

107	clist implementation	336
107.1	Allocation and initialisation	337
107.2	Predicates and conditionals	338
107.3	Retrieving data	339
107.4	Storing data	340
107.5	Mapping	341
107.6	Higher level functions	342
107.7	Stack operations	344
108	prop implementation	345
108.1	Functions	345
108.2	Predicates and conditionals	349
108.3	Mapping functions	349
109	box implementation	351
109.1	Generic boxes	351
109.2	Vertical boxes	355
109.3	Horizontal boxes	356
110	io implementation	358
110.1	Variables and constants	358
110.2	Stream management	359
110.3	Immediate writing	364
110.4	Deferred writing	365
111	Special characters for writing	366
111.1	Reading input	366
112	msg implementation	367
112.1	Variables and constants	367
112.2	Output helper functions	369
112.3	Generic functions	370
112.4	General functions	372
112.5	Redirection functions	376
112.6	Kernel-specific functions	377

113	xref implementation	380
113.1	Internal functions and variables	380
113.2	Module code	380
114	xref test file	383
115	keyval implementation	385
115.1	Module code	385
115.1.1	Variables and constants	394
115.1.2	Internal functions	395
115.1.3	Properties	403
115.1.4	Messages	407
116	Internal file functions	408
117	File operation implementation	408
118	Implementation	412
118.1	Constants	412
118.2	Variables	414
118.3	Parsing numbers	417
118.4	Internal utilities	420
118.5	Operations for fp variables	422
118.6	Transferring to other types	427
118.7	Rounding numbers	433
118.8	Unary functions	436
118.9	Basic arithmetic	438
118.10	Arithmetic for internal use	447
118.11	Trigonometric functions	454
118.12	Exponent and logarithm functions	467
118.13	Tests for special values	490
118.14	Floating-point conditionals	491
118.15	Messages	495

119Implementation	497
119.1Category code tables	497
Index	501

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 though 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*: the plain TeX `\edef`.
- f The **f** specifier stands for *full expansion*, and in contrast to *x* stops at the first non-expandable item without trying to execute it.
- T and F** For logic tests, there are the branch specifiers **T** (*true*) and **F** (*false*). Both specifiers treat the input in the same way as **n** (no change), but make the logic much easier to see.
- p The letter **p** indicates TeX *parameters*. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.
- w Finally, there is the **w** specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some arbitrary string).

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

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

- c** Constant: global parameters whose value should not be changed.
- g** Parameters whose value should only be set globally.
- l** Parameters whose value should only be set locally.

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

bool Either true or false.

box Box register.

clist Comma separated list.

dim ‘Rigid’ lengths.

int Integer-valued count 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.

num A ‘fake’ integer type using only macros. Useful for setting up allocation routines.

prop Property list.

skip ‘Rubber’ lengths.

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

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

t1 Token list variables: placeholder for a token list.

toks Token register.

1.0.1 Terminological inexactitude

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

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

<code>\ExplSyntaxOn</code>
<code>\ExplSyntaxOff</code>

`\ExplSyntaxOn ... \ExplSyntaxOff`

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

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

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

`\seq_new:N <sequence>`

When a number of variants are described, the arguments are usually illustrated only for the base function. Here, `<sequence>` 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.

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

<code>\cs_to_str:N *</code>

`\cs_to_str:N <cs>`

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 `<cs>`, shorthand for a `<control sequence>`.

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:TF *</code>

`\xetex_if_engine:TF <true code> <>false 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 `<true code>` and `<>false code>` will be shown. The two variant forms `T` and `F` take only `<true code>` and `<>false code>`, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the `expl3` modules should be defined in this way.

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

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

In some cases, the function is similar to one in L^AT_EX 2_ε or plain T_EX. In these cases, the text will include an extra ‘**T_EXhackers note**’ section:

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

The normal description text.

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

Part II

The l3names package

A systematic naming scheme for T_EX

3 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;
- defines catcode regimes for programming;
- provides settings for when the code is used in a format;
- provides tools for when the code is used as a package within a L^AT_EX 2_ε context.

4 Using the 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.

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

<code>\ExplSyntaxOn</code> <code>\ExplSyntaxOff</code>	<code>\ExplSyntaxOn <code> \ExplSyntaxOff</code>
---	--

Issues a catcode regime where spaces are ignored and colon and underscore are letters. A space character may be input with ~ instead.

<code>\ExplSyntaxNamesOn</code> <code>\ExplSyntaxNamesOff</code>	<code>\ExplSyntaxNamesOn <code> \ExplSyntaxNamesOff</code>
---	--

Issues a catcode regime where colon and underscore are letters, but spaces remain the same.

<code>\ProvidesExplPackage</code> <code>\ProvidesExplClass</code> <code>\ProvidesExplFile</code>	<code>\RequirePackage{expl3}</code> <code>\ProvidesExplPackage {<package>}</code> <code>{<date>} {<version>} {<description>}</code>
--	---

The package `l3names` (this module) provides `\ProvidesExplPackage` which is a wrapper for `\ProvidesPackage` and sets up the L^AT_EX3 catcode settings for programming automatically. Similar for the relationship between `\ProvidesExplClass` and `\ProvidesClass`. Spaces are not ignored in the arguments of these commands.

<code>\GetIdInfo</code> <code>\filename</code> <code>\filenameext</code> <code>\filedate</code> <code>\fileversion</code> <code>\filetimestamp</code> <code>\fileauthor</code> <code>\filedescription</code>	<code>\RequirePackage{l3names}</code> <code>\GetIdInfo \$Id: < cvs or svn info field> \$ {<description>}</code>
---	--

Extracts all information from a CVS or SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with `\filename` for the part of the file name leading up to the period, `\filenameext` for the extension, `\filedate` for date, `\fileversion` for version, `\filetimestamp` for the time and `\fileauthor` for the author.

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 catcodes and the L^AT_EX3 catcode 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{\filename}{\filedate}{\fileversion}{\filedescription}
```

Part III

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.

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 in the $\langle true\ arg \rangle$ or the $\langle false\ arg \rangle$. These arguments are denoted with T and F respectively. An example would be

```
\cs_if_free:cTF{abc} {\langle true code \rangle} {\langle false code \rangle}
```

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 function `\if_predicate:w` or in 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

```
\if_predicate:w \cs_if_free_p:N \l_tmpz_tl <true code> \else:
<false code> \fi:
```

or in expressions utilizing the boolean logic parser:

```
\bool_if:nTF {
  \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl
} {(true code)} {(false code)}
```

Like their branching cousins, predicate functions ensure that all underlying primitive `\else:` or `\fi:` have been removed before returning the boolean true or false values.²

For each predicate defined, a ‘predicate 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 `TEX` and `LATEX`. 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.

5.1 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_num: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_false:</code>	★	
<code>\or:</code>	★	
<code>\else:</code>	★	
<code>\fi:</code>	★	<code>\if_true: <true code> \else: <false code> \fi:</code>
<code>\reverse_if:N</code>	★	<code>\if_false: <true code> \else: <false code> \fi:</code>
		<code>\reverse_if:N <primitive conditional></code>

`\if_true:` always executes `<true code>`, while `\if_false:` always executes `<false code>`.

²If defined using the interface provided.

`\reverse_if:N` reverses any two-way primitive conditional. `\else:` and `\fi:` delimit the branches of the conditional. `\or:` is used in case switches, see `l3intexpr` 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</code> $\langle arg_1 \rangle$ $\langle arg_2 \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
------------------------------	---

`\if_meaning:w` executes $\langle true\ code \rangle$ when $\langle arg_1 \rangle$ and $\langle arg_2 \rangle$ are the same, otherwise it executes $\langle false\ code \rangle$. $\langle arg_1 \rangle$ and $\langle arg_2 \rangle$ 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</code> $\langle token_1 \rangle$ $\langle token_2 \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
<code>\if_charcode:w</code> *	<code>\if_catcode:w</code> $\langle token_1 \rangle$ $\langle token_2 \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</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 `\exp_not:N`. `\if_catcode:w` tests if the category codes of the two tokens are the same whereas `\if:w` tests if the character codes are identical. `\if_charcode:w` is an alternative name for `\if:w`.

<code>\if_predicate:w</code> *	<code>\if_predicate:w</code> $\langle predicate \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
--------------------------------	---

This function takes a predicate function and branches according to the result. (In practice this function would also accept a single boolean variable in place of the $\langle predicate \rangle$ but to make the coding clearer this should be done through `\if_bool:N`.)

<code>\if_bool:N</code> *	<code>\if_bool:N</code> $\langle boolean \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
---------------------------	--

This function takes a boolean variable and branches according to the result.

<code>\if_cs_exist:N</code> *	<code>\if_cs_exist:N</code> $\langle cs \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
<code>\if_cs_exist:w</code> *	<code>\if_cs_exist:w</code> $\langle tokens \rangle$ <code>\cs_end:</code> $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>

Check if $\langle cs \rangle$ appears in the hash table or if the control sequence that can be formed from $\langle tokens \rangle$ 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:</code> $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
<code>\if_mode_vertical:</code> *	
<code>\if_mode_math:</code> *	
<code>\if_mode_inner:</code> *	

Execute $\langle true\ code \rangle$ if currently in horizontal mode, otherwise execute $\langle false\ code \rangle$. Similar for the other functions.

5.2 Non-primitive conditionals

<code>\cs_if_eq_name_p:NN</code>

`\cs_if_eq_name_p:NN $\langle cs_1 \rangle$ $\langle cs_2 \rangle$`

Returns ‘true’ if $\langle cs_1 \rangle$ and $\langle cs_2 \rangle$ are textually the same, i.e. have the same name, otherwise it returns ‘false’.

<code>\cs_if_eq_p:NN</code>	<code>*</code>
<code>\cs_if_eq_p:cN</code>	<code>*</code>
<code>\cs_if_eq_p:Nc</code>	<code>*</code>
<code>\cs_if_eq_p:cc</code>	<code>*</code>
<code>\cs_if_eq:NNTF</code>	<code>*</code>
<code>\cs_if_eq:cNTF</code>	<code>*</code>
<code>\cs_if_eq:NcTF</code>	<code>*</code>
<code>\cs_if_eq:ccTF</code>	<code>*</code>

`\cs_if_eq_p:NNTF $\langle cs_1 \rangle$ $\langle cs_2 \rangle$`
`\cs_if_eq:NNTF $\langle cs_1 \rangle$ $\langle cs_2 \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$`

These functions check if $\langle cs_1 \rangle$ and $\langle cs_2 \rangle$ have same meaning.

<code>\cs_if_free_p:N</code>	<code>*</code>
<code>\cs_if_free_p:c</code>	<code>*</code>
<code>\cs_if_free:NTF</code>	<code>*</code>
<code>\cs_if_free:cTF</code>	<code>*</code>

`\cs_if_free_p:N $\langle cs \rangle$`
`\cs_if_free:NTF $\langle cs \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$`

Returns ‘true’ if $\langle cs \rangle$ is either undefined or equal to `\tex_relax:D` (the function that is assigned to newly created control sequences by T_EX when `\cs:w ... \cs_end:` is used). In addition to this, ‘true’ is only returned if $\langle cs \rangle$ does not have a signature equal to D, i.e., ‘do not use’ functions are not free to be redefined.

<code>\cs_if_exist_p:N</code>	<code>*</code>
<code>\cs_if_exist_p:c</code>	<code>*</code>
<code>\cs_if_exist:NTF</code>	<code>*</code>
<code>\cs_if_exist:cTF</code>	<code>*</code>

`\cs_if_exist_p:N $\langle cs \rangle$`
`\cs_if_exist:NTF $\langle cs \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$`

These functions check if $\langle cs \rangle$ exists, i.e., if $\langle cs \rangle$ is present in the hash table and is not the primitive `\tex_relax:D`.

<code>\cs_if_do_not_use_p:N</code>	<code>*</code>
------------------------------------	----------------

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

These functions check if $\langle cs \rangle$ has the arg spec D for ‘do not use’. There are no TF-type conditionals for this function as it is only used internally and not expected to be widely used. (For now, anyway.)

$\backslash\text{chk_if_free_cs:N}$ $\backslash\text{chk_if_free_cs:C}$	$\backslash\text{chk_if_free_cs:N } \langle cs \rangle$
--	--

This function checks that $\langle cs \rangle$ is $\langle free \rangle$ according to the criteria for $\backslash\text{cs_if_free_p:N}$ above. If not, an error is generated.

$\backslash\text{chk_if_exist_cs:N}$ $\backslash\text{chk_if_exist_cs:C}$	$\backslash\text{chk_if_exist_cs:N } \langle cs \rangle$
--	---

This function checks that $\langle cs \rangle$ is defined. If it is not an error is generated.

$\backslash\text{str_if_eq_p:nn } \star$ $\backslash\text{str_if_eq_p:Vn } \star$ $\backslash\text{str_if_eq_p:on } \star$ $\backslash\text{str_if_eq_p:no } \star$ $\backslash\text{str_if_eq_p:nV } \star$ $\backslash\text{str_if_eq_p:VV } \star$ $\backslash\text{str_if_eq_p:xx } \star$ $\backslash\text{str_if_eq:nnTF } \star$ $\backslash\text{str_if_eq:VnTF } \star$ $\backslash\text{str_if_eq:onTF } \star$ $\backslash\text{str_if_eq:noTF } \star$ $\backslash\text{str_if_eq:nVTF } \star$ $\backslash\text{str_if_eq:VVTF } \star$ $\backslash\text{str_if_eq:xxTF } \star$	$\backslash\text{str_if_eq_p:nn } \{ \langle tl_1 \rangle \} \{ \langle tl_2 \rangle \}$ $\backslash\text{str_if_eq:nnTF } \{ \langle tl_1 \rangle \} \{ \langle tl_2 \rangle \} \{ \langle true \text{ code} \rangle \} \{ \langle false \text{ code} \rangle \}$
---	--

Compares the two $\langle token \text{ lists} \rangle$ on a character by character basis, and is **true** if the two lists contain the same characters in the same order. Thus for example

$\backslash\text{str_if_eq_p:xx } \{ \text{ abc } \} \{ \backslash\text{tl_to_str:n } \{ \text{ abc } \} \}$

is logically **true**. The branching versions then leave either $\langle true \text{ code} \rangle$ or $\langle false \text{ code} \rangle$ in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version. All versions of these functions are fully expandable (including those involving an x-type expansion).

$\backslash\text{c_true_bool}$ $\backslash\text{c_false_bool}$

Constants that represent ‘true’ or ‘false’, respectively. Used to implement predicates.

6 Control sequences

<code>\cs:w</code>	★
<code>\cs_end:</code>	★

`\cs:w <tokens> \cs_end:`

This is the \TeX internal way of generating a control sequence from some token list. $\langle tokens \rangle$ get expanded and must ultimately result in a sequence of characters.

\TeX hackers note: These functions are the primitives `\csname` and `\endcsname`. `\cs:w` is considered weird because it expands tokens until it reaches `\cs_end:`.

<code>\cs_show:N</code>	★
<code>\cs_show:c</code>	★

`\cs_show:N <cs>`
`\cs_show:c {<arg>}`

This function shows in the console output the *meaning* of the control sequence $\langle cs \rangle$ or that created by $\langle arg \rangle$.

\TeX hackers note: This is \TeX 's `\show` and associated csname version of it.

<code>\cs_meaning:N</code>	★
<code>\cs_meaning:c</code>	★

`\cs_meaning:N <cs>`
`\cs_meaning:c {<arg>}`

This function expands to the *meaning* of the control sequence $\langle cs \rangle$ or that created by $\langle arg \rangle$.

\TeX hackers note: This is \TeX 's `\meaning` and associated csname version of it.

7 Selecting and discarding tokens from the input stream

The conditional processing cannot be implemented without being able to gobble and select which tokens to use from the input stream.

<code>\use:n</code>	★
<code>\use:nn</code>	★
<code>\use:nnn</code>	★
<code>\use:nnnn</code>	★

`\use:n {<arg>}`

Functions that returns all of their arguments to the input stream after removing the surrounding braces around each argument.

T_EXhackers note: `\use:n` is L^AT_EX 2_ε's `\@firstofone/\@iden`.

<code>\use:c</code> ★	<code>\use:c</code> $\{\langle cs \rangle\}$
-----------------------	--

Function that returns to the input stream the control sequence created from its argument. Requires two expansions before a control sequence is returned.

T_EXhackers note: `\use:c` is L^AT_EX 2_ε's `\@nameuse`.

<code>\use:x</code>	<code>\use:x</code> $\{\langle expandable\ tokens \rangle\}$
---------------------	--

Function that fully expands its argument before passing it to the input stream. Contents of the argument must be fully expandable.

T_EXhackers note: LuaT_EX provides `\expanded` which performs this operation in an expandable manner, but we cannot assume this behaviour on all platforms yet.

<code>\use_none:n</code>	★	
<code>\use_none:nn</code>	★	
<code>\use_none:nnn</code>	★	
<code>\use_none:nnnn</code>	★	
<code>\use_none:nnnnn</code>	★	
<code>\use_none:nnnnnn</code>	★	
<code>\use_none:nnnnnnn</code>	★	
<code>\use_none:nnnnnnnn</code>	★	<code>\use_none:n</code> $\{\langle arg_1 \rangle\}$
<code>\use_none:nnnnnnnnn</code>	★	<code>\use_none:nn</code> $\{\langle arg_1 \rangle\}$ $\{\langle arg_2 \rangle\}$

These functions gobble the tokens or brace groups from the input stream.

T_EXhackers note: `\use_none:n`, `\use_none:nn`, `\use_none:nnnn` are L^AT_EX 2_ε's `\@gobble`, `\@gobbletwo`, and `\@gobblefour`.

<code>\use_i:nn</code> ★	
<code>\use_ii:nn</code> ★	<code>\use_i:nn</code> $\{\langle code_1 \rangle\}$ $\{\langle code_2 \rangle\}$

Functions that execute the first or second argument respectively, after removing the surrounding braces. Primarily used to implement conditionals.

T_EXhackers note: These are L^AT_EX 2_ε's `\@firstoftwo` and `\@secondoftwo`, respectively.

<code>\use_i:nnn</code>	<code>*</code>
<code>\use_ii:nnn</code>	<code>*</code>
<code>\use_iii:nnn</code>	<code>*</code>

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

Functions that pick up one of three arguments and execute them after removing the surrounding braces.

TeXhackers note: L^AT_EX 2_ε has only `\@thirdofthree`.

<code>\use_i:nnnn</code>	<code>*</code>
<code>\use_ii:nnnn</code>	<code>*</code>
<code>\use_iii:nnnn</code>	<code>*</code>
<code>\use_iv:nnnn</code>	<code>*</code>

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

Functions that pick up one of four arguments and execute them after removing the surrounding braces.

7.1 Extending the interface

<code>\use_i_ii:nnn</code>	<code>*</code>
----------------------------	----------------

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

This function used in the expansion module reads three arguments and returns (without braces) the first and second argument while discarding the third argument.

If you wish to select multiple arguments while discarding others, use a syntax like this. Its definition is

```
\cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
```

7.2 Selecting tokens from delimited arguments

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

<code>\use_none_delimit_by_q_nil:w</code>	<code>*</code>
<code>\use_none_delimit_by_q_stop:w</code>	<code>*</code>
<code>\use_none_delimit_by_q_recursion_stop:w</code>	<code>*</code>

`\use_none_delimit_by_q_nil:w <balanced text> \q_nil`

Gobbles *<balanced text>*. Useful in gobbling the remainder in a list structure or terminating recursion.

<code>\use_i_delimit_by_q_nil:nw</code>	<code>*</code>
<code>\use_i_delimit_by_q_stop:nw</code>	<code>*</code>
<code>\use_i_delimit_by_q_recursion_stop:nw</code>	<code>*</code>

`\use_i_delimit_by_q_nil:nw {<arg>} <balanced text> \q_`

Gobbles $\langle balanced\ text \rangle$ and executes $\langle arg \rangle$ afterwards. This can also be used to get the first item in a token list.

<code>\use_i_after_fi:nw</code>	<code>*</code>	<code>\use_i_after_fi:nw {\langle arg \rangle} \fi:</code>
<code>\use_i_after_else:nw</code>	<code>*</code>	<code>\use_i_after_else:nw {\langle arg \rangle} \else: \langle balanced\ text \rangle \fi:</code>
<code>\use_i_after_or:nw</code>	<code>*</code>	<code>\use_i_after_or:nw {\langle arg \rangle} \or: \langle balanced\ text \rangle \fi:</code>
<code>\use_i_after_orelse:nw</code>	<code>*</code>	<code>\use_i_after_orelse:nw {\langle arg \rangle} \or:/\else: \langle balanced\ text \rangle \fi:</code>

Executes $\langle arg \rangle$ after executing closing out `\fi:`. `\use_i_after_orelse:nw` can be used anywhere where `\use_i_after_else:nw` or `\use_i_after_or:nw` are used.

8 That which belongs in other modules but needs to be defined earlier

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

Expands $\langle token_2 \rangle$ once and then continues processing from $\langle token_1 \rangle$.

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

<code>\exp_not:N</code>	<code>*</code>	<code>\exp_not:N \langle token \rangle</code>
<code>\exp_not:n</code>	<code>*</code>	<code>\exp_not:n {\langle tokens \rangle}</code>

In an expanding context, this function prevents $\langle token \rangle$ or $\langle tokens \rangle$ from expanding.

T_EXhackers note: These are T_EX's `\noexpand` and ε -T_EX's `\unexpanded`, respectively.

<code>\prg_do_nothing:</code>	<code>*</code>	This is as close as we get to a null operation or no-op.
-------------------------------	----------------	--

T_EXhackers note: Definition as in L^AT_EX's `\empty` but not used for the same thing.

<code>\iow_log:x</code>	
<code>\iow_term:x</code>	
<code>\iow_shipout_x:Nn</code>	<code>\iow_log:x {\message}</code> <code>\iow_shipout_x:Nn <write_stream> {\message}</code>

Writes $\langle message \rangle$ to either to log or the terminal.

<code>\msg_kernel_bug:x</code>	<code>\msg_kernel_bug:x {\message}</code>
--------------------------------	---

Internal function for calling errors in our code.

<code>\cs_record_meaning:N</code>	Placeholder for a function to be defined by <code>l3chk</code> .
-----------------------------------	--

<code>\c_minus_one</code>	Numeric constants.
<code>\c_zero</code>	
<code>\c_sixteen</code>	

9 Defining functions

There are two types of function definitions in $\text{\LaTeX}3$: versions that check if the function name is still unused, and versions that simply make the definition. The latter are used for internal scratch functions that get new meanings all over the place.

For each type there is an additional choice to be made: Does the function to be defined contain delimited arguments? The answer in 99% of the cases is no. For this type the programmer will know the number of arguments and in most cases use the argument signature to signal this, e.g., `\foo_bar:nnn` presumably takes three arguments. We therefore also provide functions that automatically detect how many arguments are required and construct the parameter text on the fly.

A definition of a new function can be done locally and globally. Currently nearly all function definitions are done locally on top level, in other words they are global but don't show it. Therefore I think it may be better to remove the local variants in the future and declare all checked function definitions global.

\TeX hackers note: While \TeX makes all definition functions directly available to the user $\text{\LaTeX}3$ hides them very carefully to avoid the problems with definitions that are overwritten accidentally. Many functions that are in \TeX a combination of prefixes and definition functions are provided as individual functions.

A slew of functions are defined in the following sections for defining new functions. Here's a quick summary to get an idea of what's available:

`\cs_(g)(new/set)(_protected)(_nopar):(N/c)(p)(n/x)`

That stands for, respectively, the following variations:

- g** Global or local;
- new/set** Define a new function or re-define an existing one;
- protected** Prevent expansion of the function in **x** arguments;
- nopar** Restrict the argument(s) from containing `\par`;
- N/c** Either a control sequence or a ‘c_sname’;
- p** Either the a primitive T_EX argument or the number of arguments is detected from the argument signature, i.e., `\foo:nnn` is assumed to have three arguments **#1#2#3**;
- n/x** Either an unexpanded or an expanded definition.

That adds up to 128 variations (!). However, the system is very logical and only a handful will usually be required often.

9.1 Defining new functions using primitive parameter text

<code>\cs_new:Npn</code> <code>\cs_new:Npx</code> <code>\cs_new:cpn</code> <code>\cs_new:cpx</code>	<code>\cs_new:Npn <cs> <parms> {<code>}</code>
--	--

Defines a function that may contain `\par` tokens in the argument(s) when called. This is not allowed for normal functions.

<code>\cs_new_nopar:Npn</code> <code>\cs_new_nopar:Npx</code> <code>\cs_new_nopar:cpn</code> <code>\cs_new_nopar:cpx</code>	<code>\cs_new_nopar:Npn <cs> <parms> {<code>}</code>
--	--

Defines a new function, making sure that `<cs>` is unused so far. `<parms>` may consist of arbitrary parameter specification in T_EX syntax. It is under the responsibility of the programmer to name the new function according to the rules laid out in the previous section. `<code>` is either passed literally or may be subject to expansion (under the **x** variants).

<code>\cs_new_protected:Npn</code> <code>\cs_new_protected:Npx</code> <code>\cs_new_protected:cpn</code> <code>\cs_new_protected:cpx</code>	<code>\cs_new_protected:Npn <cs> <parms> {<code>}</code>
--	--

Defines a function that is both robust and may contain `\par` tokens in the argument(s) when called.

<code>\cs_new_protected_nopar:Npn</code> <code>\cs_new_protected_nopar:Npx</code> <code>\cs_new_protected_nopar:cpn</code> <code>\cs_new_protected_nopar:cpx</code>	<code>\cs_new_protected_nopar:Npn <cs> <parms> {\code}</code>
--	---

Defines a function that does not expand when inside an `x` type expansion.

9.2 Defining new functions using the signature

<code>\cs_new:Nn</code> <code>\cs_new:Nx</code> <code>\cs_new:cn</code> <code>\cs_new:cx</code>	<code>\cs_new:Nn <cs> {\code}</code>
--	--

Defines a new function, making sure that `<cs>` is unused so far. The parameter text is automatically detected from the length of the function signature. If `<cs>` is missing a colon in its name, an error is raised. It is under the responsibility of the programmer to name the new function according to the rules laid out in the previous section. `<code>` is either passed literally or may be subject to expansion (under the `x` variants).

TeXhackers note: Internally, these use TeX's `\long`. These forms are recommended for low-level definitions as experience has shown that `\par` tokens often turn up in programming situations that wouldn't have been expected.

<code>\cs_new_nopar:Nn</code> <code>\cs_new_nopar:Nx</code> <code>\cs_new_nopar:cn</code> <code>\cs_new_nopar:cx</code>	<code>\cs_new_nopar:Nn <cs> {\code}</code>
--	--

Version of the above in which `\par` is not allowed to appear within the argument(s) of the defined functions.

<code>\cs_new_protected:Nn</code> <code>\cs_new_protected:Nx</code> <code>\cs_new_protected:cn</code> <code>\cs_new_protected:cx</code>	<code>\cs_new_protected:Nn <cs> {\code}</code>
--	--

Defines a function that is both robust and may contain `\par` tokens in the argument(s) when called.

<code>\cs_new_protected_nopar:Nn</code> <code>\cs_new_protected_nopar:Nx</code> <code>\cs_new_protected_nopar:cn</code> <code>\cs_new_protected_nopar:cx</code>	<code>\cs_new_protected_nopar:Nn <cs> {<code>}</code>
--	---

Defines a function that does not expand when inside an `x` type expansion. `\par` is not allowed in the argument(s) of the defined function.

9.3 Defining functions using primitive parameter text

Besides the function definitions that check whether or not their argument is an unused function we need function definitions that overwrite currently used definitions. The following functions are provided for this purpose.

<code>\cs_set:Npn</code> <code>\cs_set:Npx</code> <code>\cs_set:cpn</code> <code>\cs_set:cpx</code>	<code>\cs_set:Npn <cs> <parms> {<code>}</code>
--	--

Like `\cs_set_nopar:Npn` but allows `\par` tokens in the arguments of the function being defined.

TeXhackers note: These are equivalent to TeX's `\long\def` and so on. These forms are recommended for low-level definitions as experience has shown that `\par` tokens often turn up in programming situations that wouldn't have been expected.

<code>\cs_gset:Npn</code> <code>\cs_gset:Npx</code> <code>\cs_gset:cpn</code> <code>\cs_gset:cpx</code>	<code>\cs_gset:Npn <cs> <parms> {<code>}</code>
--	---

Global variant of `\cs_set:Npn`.

<code>\cs_set_nopar:Npn</code> <code>\cs_set_nopar:Npx</code> <code>\cs_set_nopar:cpn</code> <code>\cs_set_nopar:cpx</code>	<code>\cs_set_nopar:Npn <cs> <parms> {<code>}</code>
--	--

Like `\cs_new_nopar:Npn` etc. but does not check the `<cs>` name.

T_EXhackers note: `\cs_set_nopar:Npn` is the L^AT_EX3 name for T_EX’s `\def` and `\cs_set_nopar:Npx` corresponds to the primitive `\edef`. The `\cs_set_nopar:cpn` function was known in L^AT_EX2 as `\@namedef`. `\cs_set_nopar:cpx` has no equivalent.

<code>\cs_gset_nopar:Npn</code> <code>\cs_gset_nopar:Npx</code> <code>\cs_gset_nopar:cpn</code> <code>\cs_gset_nopar:cpx</code>	<code>\cs_gset_nopar:Npn <cs> <parms> {\code}</code>
--	--

Like `\cs_set_nopar:Npn` but defines the `<cs>` globally.

T_EXhackers note: `\cs_gset_nopar:Npn` and `\cs_gset_nopar:Npx` are T_EX’s `\gdef` and `\xdef`.

<code>\cs_set_protected:Npn</code> <code>\cs_set_protected:Npx</code> <code>\cs_set_protected:cpn</code> <code>\cs_set_protected:cpx</code>	<code>\cs_set_protected:Npn <cs> <parms> {\code}</code>
--	---

Naturally robust macro that won’t expand in an `x` type argument. These varieties allow `\par` tokens in the arguments of the function being defined.

<code>\cs_gset_protected:Npn</code> <code>\cs_gset_protected:Npx</code> <code>\cs_gset_protected:cpn</code> <code>\cs_gset_protected:cpx</code>	<code>\cs_gset_protected:Npn <cs> <parms> {\code}</code>
--	--

Global versions of the above functions.

<code>\cs_set_protected_nopar:Npn</code> <code>\cs_set_protected_nopar:Npx</code> <code>\cs_set_protected_nopar:cpn</code> <code>\cs_set_protected_nopar:cpx</code>	<code>\cs_set_protected_nopar:Npn <cs> <parms> {\code}</code>
--	---

Naturally robust macro that won’t expand in an `x` type argument. If you want for some reason to expand it inside an `x` type expansion, prefix it with `\exp_after:wN \prg_do_nothing:`.

<code>\cs_gset_protected_nopar:Npn</code> <code>\cs_gset_protected_nopar:Npx</code> <code>\cs_gset_protected_nopar:cpn</code> <code>\cs_gset_protected_nopar:cpx</code>	<code>\cs_gset_protected_nopar:Npn <cs> <parms> {\code}</code>
--	--

Global versions of the above functions.

9.4 Defining functions using the signature (no checks)

As above but now detecting the parameter text from inspecting the signature.

<code>\cs_set:Nn</code>
<code>\cs_set:Nx</code>
<code>\cs_set:cn</code>
<code>\cs_set:cx</code>

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

Like `\cs_set_nopar:Nn` but allows `\par` tokens in the arguments of the function being defined.

<code>\cs_gset:Nn</code>
<code>\cs_gset:Nx</code>
<code>\cs_gset:cn</code>
<code>\cs_gset:cx</code>

`\cs_gset:Nn <cs> {<code>}`

Global variant of `\cs_set:Nn`.

<code>\cs_set_nopar:Nn</code>
<code>\cs_set_nopar:Nx</code>
<code>\cs_set_nopar:cn</code>
<code>\cs_set_nopar:cx</code>

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

Like `\cs_new_nopar:Nn` etc. but does not check the `<cs>` name.

<code>\cs_gset_nopar:Nn</code>
<code>\cs_gset_nopar:Nx</code>
<code>\cs_gset_nopar:cn</code>
<code>\cs_gset_nopar:cx</code>

`\cs_gset_nopar:Nn <cs> {<code>}`

Like `\cs_set_nopar:Nn` but defines the `<cs>` globally.

<code>\cs_set_protected:Nn</code>
<code>\cs_set_protected:cn</code>
<code>\cs_set_protected:Nx</code>
<code>\cs_set_protected:cx</code>

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

Naturally robust macro that won't expand in an `x` type argument. These varieties also allow `\par` tokens in the arguments of the function being defined.

<code>\cs_gset_protected:Nn</code>
<code>\cs_gset_protected:cn</code>
<code>\cs_gset_protected:Nx</code>
<code>\cs_gset_protected:cx</code>

`\cs_gset_protected:Nn <cs> {<code>}`

Global versions of the above functions.

<code>\cs_set_protected_nopar:Nn</code> <code>\cs_set_protected_nopar:cn</code> <code>\cs_set_protected_nopar:Nx</code> <code>\cs_set_protected_nopar:cx</code>	<code>\cs_set_protected_nopar:Nn <cs> {<code>}</code>
--	---

Naturally robust macro that won't expand in an `x` type argument. This also comes as a `long` version. If you for some reason want to expand it inside an `x` type expansion, prefix it with `\exp_after:wN \prg_do_nothing:.`

<code>\cs_gset_protected_nopar:Nn</code> <code>\cs_gset_protected_nopar:cn</code> <code>\cs_gset_protected_nopar:Nx</code> <code>\cs_gset_protected_nopar:cx</code>	<code>\cs_gset_protected_nopar:Nn <cs> {<code>}</code>
--	--

Global versions of the above functions.

9.5 Undefined functions

<code>\cs_undefine:N</code> <code>\cs_undefine:c</code> <code>\cs_gundefine:N</code> <code>\cs_gundefine:c</code>	<code>\cs_gundefine:N <cs></code>
--	---

Undefines the control sequence locally or globally. In a global context, this is useful for reclaiming a small amount of memory but shouldn't often be needed for this purpose. In a local context, this can be useful if you need to clear a definition before applying a short-term modification to something.

9.6 Copying function definitions

<code>\cs_new_eq:NN</code> <code>\cs_new_eq:cN</code> <code>\cs_new_eq:Nc</code> <code>\cs_new_eq:cc</code>	<code>\cs_new_eq:NN <cs₁₂</code>
--	---

Gives the function `<cs1 locally or globally the current meaning of <cs2. If <cs1 already`

exists then an error is called.

<code>\cs_set_eq:NN</code>
<code>\cs_set_eq:cN</code>
<code>\cs_set_eq:Nc</code>
<code>\cs_set_eq:cc</code>
<code>\cs_gset_eq:NN</code>
<code>\cs_gset_eq:cN</code>
<code>\cs_gset_eq:Nc</code>
<code>\cs_gset_eq:cc</code>

`\cs_set_eq:cN <cs1> <cs2>`

Gives the function $\langle cs_1 \rangle$ the current meaning of $\langle cs_2 \rangle$. Again, we may always do this globally.

<code>\cs_set_eq:NwN</code>

`\cs_set_eq:NwN <cs1> <cs2>`
`\cs_set_eq:NwN <cs1> = <cs2>`

These functions assign the meaning of $\langle cs_2 \rangle$ locally or globally to the function $\langle cs_1 \rangle$. Because the \TeX primitive operation is being used which may have an equal sign and (a certain number of) spaces between $\langle cs_1 \rangle$ and $\langle cs_2 \rangle$ the name contains a `w`. (Not happy about this convention!).

\TeX hackers note: `\cs_set_eq:NwN` is the \LaTeX 3 name for \TeX 's `\let`.

9.7 Internal functions

<code>\pref_global:D</code>
<code>\pref_long:D</code>
<code>\pref_protected:D</code>

`\pref_global:D \cs_set_nopar:Npn`

Prefix functions that can be used in front of some definition functions (namely ...). The result of prefixing a function definition with `\pref_global:D` makes the definition global, `\pref_long:D` change the argument scanning mechanism so that it allows `\par` tokens in the argument of the prefixed function, and `\pref_protected:D` makes the definition robust in `\writes` etc.

None of these internal functions should be used by a programmer since the necessary combinations are all available as separate function, e.g., `\cs_set:Npn` is internally implemented as `\pref_long:D \cs_set_nopar:Npn`.

\TeX hackers note: These prefixes are the primitives `\global`, `\long`, and `\protected`. The `\outer` prefix isn't used at all within \LaTeX 3 because ... (it causes more hassle than it's worth? It's never proved useful in any meaningful way?)

10 The innards of a function

`\cs_to_str:N *` `\cs_to_str:N <cs>`

This function returns the name of `<cs>` as a sequence of letters with the escape character removed.

`\token_to_str:N *`
`\token_to_str:c *` `\token_to_str:N <arg>`

This function return the name of `<arg>` as a sequence of letters including the escape character.

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

`\token_to_meaning:N *` `\token_to_meaning:N <arg>`

This function returns the type and definition of `<arg>` as a sequence of letters.

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

`\cs_get_function_name:N *`
`\cs_get_function_signature:N *` `\cs_get_function_name:N \<fn>:<args>`

The **name** variant strips off the leading escape character and the trailing argument specification (including the colon) to return `<fn>`. The **signature** variants does the same but returns the signature `<args>` instead.

`\cs_split_function:NN *` `\cs_split_function:NN \<fn>:<args> <post process>`

Strips off the leading escape character, splits off the signature without the colon, informs whether or not a colon was present and then prefixes these results with `<post process>`, i.e., `<post process>{\<name>}{\<signature>}{\<true>/\<false>}`. For example, `\cs_get_function_name:N` is nothing more than `\cs_split_function:NN \<fn>:<args>` `\use_i:nnn`.

`\cs_get_arg_count_from_signature:N *` `\cs_get_arg_count_from_signature:N \<fn>:<args>`

Returns the number of chars in `<args>`, signifying the number of arguments that the function uses.

Other functions regarding arbitrary tokens can be found in the `l3token` module.

11 Grouping and scanning

`\scan_stop:` `\scan_stop:`

This function stops T_EX's scanning ahead when ending a number.

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

`\group_begin:`

`\group_end:`

`\group_begin: <...> \group_end:`

Encloses `<...>` inside a group.

T_EXhackers note: These are the T_EX primitives `\begingroup` and `\endgroup` renamed.

`\group_execute_after:N`

`\group_execute_after:N <token>`

Adds `<token>` to the list of tokens to be inserted after the current group ends (through an explicit or implicit `\group_end:`).

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

12 Checking the engine

`\xetex_if_engine:TF *`

`\xetex_if_engine:TF {<true code>} {<false code>}`

This function detects if we're running a XeT_EX-based format.

`\luatex_if_engine:TF *`

`\luatex_if_engine:TF {<true code>} {<false code>}`

This function detects if we're running a LuaT_EX-based format.

`\c_xetex_is_engine_bool`

`\c_luatex_is_engine_bool`

Boolean variables used for the above functions.

Part IV

The `l3expan` package

Controlling Expansion of Function Arguments

13 Brief overview

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 `LATEX3` 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.

14 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 the third argument gets expanded once. If `\seq_gpush:No` wouldn't be defined the example above 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.

14.1 Methods for defining variants

<code>\cs_generate_variant:Nn</code>	<code>\cs_generate_variant:Nn <parent control sequence> {<variant argument specifier>}</code>
--------------------------------------	---

The *<parent control sequence>* is first separated into the *<base name>* and *<original>* argument specifier. The *<variant>* is then used to modify this by replacing the beginning of the *<original>* with the *<variant>*. Thus the *<variant>* must be no longer than the *<original>* argument specifier. This new specifier is used to create a modified function which will expand its arguments as required. 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 }
\cs_generate_variant:Nn \foo:Nn { cV }
```

would generate the functions `\foo:NV` and `\foo:cV` in the same way. `\cs_generate_variant:Nn` can only be applied if the *<parent control sequence>* is already defined. If the *<parent control sequence>* is protected then the new sequence will also be protected. The variants are generated globally.

Internal functions

<code>\cs_generate_internal_variant:n</code>	<code>\cs_generate_internal_variant:n {<args>}</code>
--	---

Defines the appropriate `\exp_args:N<args>` function, if necessary, to perform the expansion control specified by *<args>*.

15 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 very fast.
- In general `n`, `x`, and `o` (if not in the last position) will need special processing which is not fast and not expandable, i.e., functions of this type may not work correctly in arguments that are itself subject to `x` expansion. Therefore it is best to use the “expandable” functions (i.e., those that contain only `c`, `N`, `o` or `f` in the last position) whenever possible.

The `V` type returns the value of a register, which can be one of `tl`, `num`, `int`, `skip`, `dim`, `toks`, or built-in `TEX` registers. The `v` type is the same except it first creates a control sequence out of its argument before returning the value. 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 `\aaa` equal to the control sequence stemming from turning `b \l_tmpa_tl b` into a control sequence. Furthermore we want to store the execution of it in a `<toks>` register. In this example we assume `\l_tmpa_tl` contains the text string `lur`. The straight forward approach is

```
\toks_set:No \l_tmpa_toks {\cs_set_eq:Nc \aaa {b \l_tmpa_tl b}}
```

Unfortunately this only puts `\exp_args:NNc \cs_set_eq:NN \aaa {b \l_tmpa_tl b}` into `\l_tmpa_toks` and not `\cs_set_eq:NwN \aaa = \blurb` as we probably wanted. Using `\toks_set:Nx` is not an option as that will die horribly. Instead we can do a

```
\toks_set:Nf \l_tmpa_toks {\cs_set_eq:Nc \aaa {b \l_tmpa_tl b}}
```

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

```
\cs_set_nopar:Npn \toks_set:Nf {\exp_args:NNf \toks_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!

16 Manipulating the first argument

`\exp_args:No` ★ `\exp_args:No` $\langle\textit{funct}\rangle$ $\langle\textit{arg}_1\rangle$ $\langle\textit{arg}_2\rangle$...

The first argument of $\langle\textit{funct}\rangle$ (i.e., $\langle\textit{arg}_1\rangle$) is expanded once, the result is surrounded by braces and passed to $\langle\textit{funct}\rangle$. $\langle\textit{funct}\rangle$ may have more than one argument—all others are passed unchanged.

`\exp_args:Nc` ★
`\exp_args:cc` ★ `\exp_args:Nc` $\langle\textit{funct}\rangle$ $\langle\textit{arg}_1\rangle$ $\langle\textit{arg}_2\rangle$...

The first argument of $\langle\textit{funct}\rangle$ (i.e., $\langle\textit{arg}_1\rangle$) is expanded until only characters remain. (An internal error occurs if something else is the result of this expansion.) Then the result is turned into a control sequence and passed to $\langle\textit{funct}\rangle$ as the first argument. $\langle\textit{funct}\rangle$ may have more than one argument—all others are passed unchanged.

In the `:cc` variant, the $\langle\textit{funct}\rangle$ control sequence itself is constructed (with the same process as described above) before $\langle\textit{arg}_1\rangle$ is turned into a control sequence and passed as its argument.

`\exp_args:NV` ★ `\exp_args:NV` $\langle\textit{funct}\rangle$ $\langle\textit{register}\rangle$

The first argument of $\langle\textit{funct}\rangle$ (i.e., $\langle\textit{register}\rangle$) is expanded to its value. By value we mean a number stored in an `int` or `num` register, the length value of a `dim`, `skip` or `muskip` register, the contents of a `toks` register or the unexpanded contents of a `tl var.` register. The value is passed onto $\langle\textit{funct}\rangle$ in braces.

`\exp_args:Nv` ★ `\exp_args:Nv` $\langle\textit{funct}\rangle$ $\{\langle\textit{register}\rangle\}$

Like the `V` type except the register is given by a list of characters from which a control sequence name is generated.

`\exp_args:Nx` `\exp_args:Nx` $\langle\textit{funct}\rangle$ $\langle\textit{arg}_1\rangle$ $\langle\textit{arg}_2\rangle$...

The first argument of $\langle\textit{funct}\rangle$ (i.e., $\langle\textit{arg}_1\rangle$) is fully expanded until only unexpandable tokens remain, the result is surrounded by braces and passed to $\langle\textit{funct}\rangle$. $\langle\textit{funct}\rangle$ may

have more than one argument—all others are passed unchanged. As mentioned before, this type of function is relatively slow.

<code>\exp_args:Nf</code> ★

`\exp_args:Nf` $\langle funct \rangle$ $\langle arg_1 \rangle$ $\langle arg_2 \rangle$...

The first argument of $\langle funct \rangle$ (i.e., $\langle arg_1 \rangle$) undergoes full expansion until the first unexpandable token is encountered, the result is surrounded by braces and passed to $\langle funct \rangle$. $\langle funct \rangle$ may have more than one argument—all others are passed unchanged. Beware of its special behavior as explained above.

17 Manipulating two arguments

<code>\exp_args:NNx</code>
<code>\exp_args:Nnx</code>
<code>\exp_args:Ncx</code>
<code>\exp_args:Nox</code>
<code>\exp_args:Nxo</code>
<code>\exp_args:Nxx</code>

`\exp_args:Nnx` $\langle funct \rangle$ $\langle arg_1 \rangle$ $\langle arg_2 \rangle$...

The above functions all manipulate the first two arguments of $\langle funct \rangle$. They are all slow and non-expandable.

<code>\exp_args:NNo</code> ★
<code>\exp_args:NNc</code> ★
<code>\exp_args:NNv</code> ★
<code>\exp_args:NNV</code> ★
<code>\exp_args:NNf</code> ★
<code>\exp_args:Nno</code> ★
<code>\exp_args:NnV</code> ★
<code>\exp_args:Nnf</code> ★
<code>\exp_args:Noo</code> ★
<code>\exp_args:Noc</code> ★
<code>\exp_args:Nco</code> ★
<code>\exp_args:Ncf</code> ★
<code>\exp_args:Ncc</code> ★
<code>\exp_args:Nff</code> ★
<code>\exp_args:Nfo</code> ★
<code>\exp_args:NVV</code> ★

`\exp_args:NNo` $\langle funct \rangle$ $\langle arg_1 \rangle$ $\langle arg_2 \rangle$...

These are the fast and expandable functions for the first two arguments.

18 Manipulating three arguments

So far not all possible functions are provided and even the selection below may be reduced in the future as far as the non-expandable functions are concerned.

<code>\exp_args:NNnx</code>
<code>\exp_args:NNox</code>
<code>\exp_args:Nnnx</code>
<code>\exp_args:Nnox</code>
<code>\exp_args:Noox</code>
<code>\exp_args:Ncnx</code>
<code>\exp_args:Nccx</code>

`\exp_args:Nnnx <funct> <arg1> <arg2> <arg3> ...`

All the above functions are non-expandable.

<code>\exp_args:NNNo</code>	<code>*</code>
<code>\exp_args:NNNV</code>	<code>*</code>
<code>\exp_args:NNoo</code>	<code>*</code>
<code>\exp_args:NNno</code>	<code>*</code>
<code>\exp_args:Nnno</code>	<code>*</code>
<code>\exp_args:Nnnc</code>	<code>*</code>
<code>\exp_args:Nooo</code>	<code>*</code>
<code>\exp_args:Nccc</code>	<code>*</code>
<code>\exp_args:NcNc</code>	<code>*</code>
<code>\exp_args:NcNo</code>	<code>*</code>
<code>\exp_args:Ncco</code>	<code>*</code>

`\exp_args:NNoo <funct> <arg1> <arg2> <arg3> ...`

These are the fast and expandable functions for the first three arguments.

19 Preventing expansion

<code>\exp_not:N</code>
<code>\exp_not:c</code>
<code>\exp_not:n</code>

`\exp_not:N <token>`
`\exp_not:n {<token list>}`

This function will prohibit the expansion of `<token>` in situation where `<token>` would otherwise be replaced by its definition, e.g., inside an argument that is handled by the `x` convention.

TeXhackers note: `\exp_not:N` is the primitive `\noexpand` renamed and `\exp_not:n` is the ε -TeX primitive `\unexpanded`.

<code>\exp_not:o</code>
<code>\exp_not:f</code>

`\exp_not:o {⟨token list⟩}`

Same as `\exp_not:n` except `⟨token list⟩` is expanded once for the `o` type and for the `f` type the token list is expanded until an unexpandable token is found, and the result of these expansions is then prohibited from being expanded further.

<code>\exp_not:V</code>
<code>\exp_not:v</code>

`\exp_not:V ⟨register⟩`
`\exp_not:v {⟨token list⟩}`

The value of `⟨register⟩` is retrieved and then passed on to `\exp_not:n` which will prohibit further expansion. The `v` type first creates a control sequence from `⟨token list⟩` but is otherwise identical to `V`.

<code>\exp_stop_f:</code>

`⟨f expansion⟩ ... \exp_stop_f:`

This function stops an `f` type expansion. An example use is one such as

```
\tl_set:Nf \l_tmpa_tl {
  \if_case:w \l_tmpa_int
    \or:   \use_i_after_orelse:nw {\exp_stop_f: \textbullet}
    \or:   \use_i_after_orelse:nw {\exp_stop_f: \textendash}
    \else: \use_i_after_fi:nw      {\exp_stop_f: else-item}
  \fi:
}
```

This ensures the expansion is stopped right after finishing the conditional but without expanding `\textbullet` etc.

T_EXhackers note: This function is a space token but it is better to distinguish this expansion stopping token from a desired space token when writing code.

20 Unbraced expansion

<code>\exp_last_unbraced:Nf</code>
<code>\exp_last_unbraced:NV</code>
<code>\exp_last_unbraced:No</code>
<code>\exp_last_unbraced:Nv</code>
<code>\exp_last_unbraced:NcV</code>
<code>\exp_last_unbraced:NNV</code>
<code>\exp_last_unbraced:NNo</code>
<code>\exp_last_unbraced:NNV</code>
<code>\exp_last_unbraced:NNNo</code>

`\exp_last_unbraced:NV ⟨token⟩ ⟨variable name⟩`

There are a small number of occasions where the last argument in an expansion run must be expanded unbraced. These functions should only be used inside functions, *not* for creating variants.

Part V

The l3prg package

Program control structures

21 Conditionals and logical operations

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 $\langle true \rangle$ and $\langle false \rangle$ but other states are possible, say an $\langle error \rangle$ state for erroneous input, e.g., text as input in a function comparing integers.

L^AT_EX3 has two primary forms of conditional flow processing based on these states. One type is predicate functions that turn the returned state into a boolean $\langle true \rangle$ or $\langle false \rangle$. 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 $\langle true \rangle$ or $\langle false \rangle$ values to be used in testing with `\if_predicate:w` or in functions to be described below. The other type 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 $\langle true \rangle$ or $\langle false \rangle$ depending on the result. Important to note here is that the arguments are executed after exiting the underlying `\if...\fi:` structure

22 Defining a set of conditional functions

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

These functions exit conditional processing when used in con-

junction with the generating functions listed below.

<pre> \prg_set_conditional:Nnn \prg_set_conditional:Npnn \prg_new_conditional:Nnn \prg_new_conditional:Npnn \prg_set_protected_conditional:Nnn \prg_set_protected_conditional:Npnn \prg_new_protected_conditional:Nnn \prg_new_protected_conditional:Npnn \prg_set_eq_conditional:NNn \prg_new_eq_conditional:NNn </pre>	<pre> \prg_set_conditional:Nnn <test> <conds> <code> \prg_set_conditional:Npnn <test> <param> <conds> <code> </pre>
--	---

This defines a conditional *<base function>* which upon evaluation using `\prg_return_true:` and `\prg_return_false:` to finish branches, returns a state. Currently the states are either *<true>* or *<false>* although this can change as more states may be introduced, say an *<error>* state. *<conds>* is a comma separated list possibly consisting of *p* for denoting a predicate function returning the boolean *<true>* or *<false>* values and *TF*, *T* and *F* for the functions that act on the tokens following in the input stream. The `:Nnn` form implicitly determines the number of arguments from the function being defined whereas the `:Npnn` form expects a primitive parameter text.

An example can easily clarify matters here:

```

\prg_set_conditional:Nnn \foo_if_bar:NN {p,TF,T} {
  \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`, `\foo_if_bar:NNT` but not `\foo_if_bar:NNF` (because *F* is missing from the *<conds>* 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 an error if that branch is executed.

23 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 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:c</code>	<code>\bool_new:N <bool></code>
--	---------------------------------------

Define a new boolean variable. The initial value is `<false>`. A boolean is actually just either `\c_true_bool` or `\c_false_bool`.

<code>\bool_set_true:N</code> <code>\bool_set_true:c</code> <code>\bool_set_false:N</code> <code>\bool_set_false:c</code> <code>\bool_gset_true:N</code> <code>\bool_gset_true:c</code> <code>\bool_gset_false:N</code> <code>\bool_gset_false:c</code>	<code>\bool_gset_false:N <bool></code>
--	--

Set `<bool>` either `<true>` or `<false>`. We can also do this globally.

<code>\bool_set_eq:NN</code> <code>\bool_set_eq:Nc</code> <code>\bool_set_eq:cN</code> <code>\bool_set_eq:cc</code> <code>\bool_gset_eq:NN</code> <code>\bool_gset_eq:Nc</code> <code>\bool_gset_eq:cN</code> <code>\bool_gset_eq:cc</code>	<code>\bool_set_eq:NN <bool₁> <bool₂></code>
--	--

Set $\langle bool_1 \rangle$ equal to the value of $\langle bool_2 \rangle$.

<code>\bool_if_p:N *</code>	
<code>\bool_if:NTF *</code>	
<code>\bool_if_p:c *</code>	<code>\bool_if:NTF <bool> {<true>} {<false>}</code>
<code>\bool_if:cTF *</code>	<code>\bool_if_p:N <bool></code>

Test the truth value of $\langle bool \rangle$ and execute the $\langle true \rangle$ or $\langle false \rangle$ code. `\bool_if_p:N` is a predicate function for use in `\if_predicate:w` tests or `\bool_if:nTF`-type functions described below.

<code>\bool_while_do:Nn</code>	
<code>\bool_while_do:cn</code>	
<code>\bool_until_do:Nn</code>	
<code>\bool_until_do:cn</code>	
<code>\bool_do_while:Nn</code>	
<code>\bool_do_while:cn</code>	
<code>\bool_do_until:Nn</code>	<code>\bool_while_do:Nn <bool> {<code>}</code>
<code>\bool_do_until:cn</code>	<code>\bool_until_do:Nn <bool> {<code>}</code>

The ‘while’ versions execute $\langle code \rangle$ as long as the boolean is true and the ‘until’ versions execute $\langle code \rangle$ as long as the boolean is false. The `while_do` functions execute the body after testing the boolean and the `do_while` functions executes the body first and then tests the boolean.

24 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 *boolean expressions*.

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 `!`. 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 anymore, the remaining

tests within the current group are skipped.

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

The functions evaluate the truth value of $\langle \text{boolean expression} \rangle$ where each predicate is separated by `&&` or `||` denoting logical ‘And’ and ‘Or’ functions. `(` and `)` denote grouping of sub-expressions while `!` is used to as a prefix to either negate a single expression or a group. Hence

```

\bool_if_p:n{
  \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})
}

```

from above returns $\langle \text{true} \rangle$.

Logical operators take higher precedence the later in the predicate they appear. “ $\langle x \rangle || \langle y \rangle \&\& \langle z \rangle$ ” is interpreted as the equivalent of “ $\langle x \rangle$ OR [$\langle y \rangle$ AND $\langle z \rangle$]” (but now we have grouping you shouldn’t write this sort of thing, anyway).

<code>\bool_not_p:n *</code>	<code>\bool_not_p:n {<boolean expression>}</code>
------------------------------	---

Longhand for writing `!(<boolean expression>)` within a boolean expression. Might not stick around.

<code>\bool_xor_p:nn *</code>	<code>\bool_xor_p:nn {<boolean expression>} {<boolean expression>}</code>
-------------------------------	---

Implements an ‘exclusive or’ operation between two boolean expressions. There is no infix operation for this.

<code>\bool_set:Nn</code> <code>\bool_set:cn</code> <code>\bool_gset:Nn</code> <code>\bool_gset:cn</code>	<code>\bool_set:Nn <bool> {<boolean expression>}</code>
--	---

Sets $\langle \text{bool} \rangle$ to the logical outcome of evaluating $\langle \text{boolean expression} \rangle$.

25 Case switches

```

\prg_case_int:nnn {<integer expr>} {
  {<integer expr1>} {<code1>}}
  {<integer expr2>} {<code2>}}
  ...
  {<integer exprn>} {<coden>}}
\prg_case_int:nnn * } {<else case>}}

```

This function evaluates the first $\langle integer\ expr \rangle$ and then compares it to the values found in the list. Thus the expression

```

\prg_case_int:nnn{2*5}{
  {5}{Small} {4+6}{Medium} {-2*10}{Negative}
}{0ther}

```

evaluates first the term to look for and then tries to find this value in the list of values. If the value is found, the code on its right is executed after removing the remainder of the list. If the value is not found, the $\langle else\ case \rangle$ is executed. The example above will return “Medium”.

The function is expandable and is written in such a way that **f** style expansion can take place cleanly, i.e., no tokens from within the function are left over.

```

\prg_case_int:nnn {<dim expr>} {
  {<dim expr1>} {<code1>}}
  {<dim expr2>} {<code2>}}
  ...
  {<dim exprn>} {<coden>}}
\prg_case_dim:nnn * } {<else case>}}

```

This function works just like `\prg_case_int:nnn` except it works for $\langle dim \rangle$ registers.

```

\prg_case_str:nnn {<string>} {
  {<string1>} {<code1>}}
  {<string2>} {<code2>}}
  ...
  {<stringn>} {<coden>}}
\prg_case_str:nnn * } {<else case>}}

```

This function works just like `\prg_case_int:nnn` except it compares strings. Each string is evaluated fully using **x** style expansion.

The function is expandable³ and is written in such a way that **f** style expansion can take place cleanly, i.e., no tokens from within the function are left over.

```

\prg_case_tl:Nnn <tl var.> {
  <tl var.1> {<code1>} <tl var.2> {<code2>} ... <tl var.n>
  {<coden>}}
\prg_case_tl:Nnn * } {<else case>}}

```

³Provided you use pdfTeX v1.30 or later

This function works just like `\prg_case_int:nnn` except it compares token list variables. The function is expandable⁴ and is written in such a way that `f` style expansion can take place cleanly, i.e., no tokens from within the function are left over.

26 Generic loops

<code>\bool_while_do:nn</code> <code>\bool_until_do:nn</code> <code>\bool_do_while:nn</code> <code>\bool_do_until:nn</code>	<code>\bool_while_do:nn {<boolean expression>} {<code>}</code> <code>\bool_until_do:nn {<boolean expression>} {<code>}</code>
--	--

The ‘while’ versions execute the code as long as *<boolean expression>* is true and the ‘until’ versions execute *<code>* as long as *<boolean expression>* is false. The `while_do` functions execute the body after testing the boolean and the `do_while` functions executes the body first and then tests the boolean.

27 Choosing modes

<code>\mode_if_vertical_p: *</code> <code>\mode_if_vertical:TF *</code>	<code>\mode_if_vertical:TF {<true code>} {<false code>}</code>
--	--

Determines if T_EX is in vertical mode or not and executes either *<true code>* or *<false code>* accordingly.

<code>\mode_if_horizontal_p: *</code> <code>\mode_if_horizontal:TF *</code>	<code>\mode_if_horizontal:TF {<true code>} {<false code>}</code>
--	--

Determines if T_EX is in horizontal mode or not and executes either *<true code>* or *<false code>* accordingly.

<code>\mode_if_inner_p: *</code> <code>\mode_if_inner:TF *</code>	<code>\mode_if_inner:TF {<true code>} {<false code>}</code>
--	---

Determines if T_EX is in inner mode or not and executes either *<true code>* or *<false code>* accordingly.

<code>\mode_if_math_p: *</code> <code>\mode_if_math:TF *</code>	<code>\mode_if_math:TF {<true code>} {<false code>}</code>
--	--

Determines if T_EX is in math mode or not and executes either *<true code>* or *<false code>* accordingly.

⁴Provided you use pdfT_EX v1.30 or later

T_EXhackers note: This version will choose the right branch even at the beginning of an alignment cell.

28 Alignment safe grouping and scanning

<code>\scan_align_safe_stop:</code>

`\scan_align_safe_stop:`

This function gets T_EX on the right track inside an alignment cell but without destroying any kerning.

<code>\group_align_safe_begin:</code>
<code>\group_align_safe_end:</code>

`\group_align_safe_begin: <...> \group_align_safe_end:`

Encloses <...> inside a group but is safe inside an alignment cell. See the implementation of `\peek_token_generic:NNTF` for an application.

29 Producing n copies

There are often several different requirements for producing multiple copies of something. Sometimes one might want to produce a number of identical copies of a sequence of tokens whereas at other times the goal is to simulate a for loop as known from most real programming languages.

<code>\prg_replicate:nn *</code>

`\prg_replicate:nn <{number}> <{arg}>`

Creates <number> copies of <arg>. Note that it is expandable.

<code>\prg_stepwise_function:nnnN *</code>
--

`\prg_stepwise_function:nnnN <{start}> <{step}> <{end}> <function>`

This function performs <action> once for each step starting at <start> and ending once <end> is passed. <function> is placed directly in front of a brace group holding the current number so it should usually be a function taking one argument.

<code>\prg_stepwise_inline:nnnn</code>
--

`\prg_stepwise_inline:nnnn <{start}> <{step}> <{end}> <{action}>`

Same as `\prg_stepwise_function:nnnN` except here <action> is performed each time

with `##1` as a placeholder for the number currently being tested. This function is not expandable and it is nestable.

<code>\prg_stepwise_variable:nnnNn</code>	<code>\prg_stepwise_variable:nnnn {<start>} {<step>} {<end>} <temp-var> {<action>}</code>
---	---

Same as `\prg_stepwise_inline:nnnn` except here the current value is stored in `<temp-var>` and the programmer can use it in `<action>`. This function is not expandable.

30 Sorting

<code>\prg_quicksort:n</code>	<code>\prg_quicksort:n { {<item₁>} {<item₂>} ... {<item_n>} }</code>
-------------------------------	--

Performs a Quicksort on the token list. The comparisons are performed by the function `\prg_quicksort_compare:nnTF` which is up to the programmer to define. When the sorting process is over, all items are given as argument to the function `\prg_quicksort_function:n` which the programmer also controls.

<code>\prg_quicksort_function:n</code> <code>\prg_quicksort_compare:nnTF</code>	<code>\prg_quicksort_function:n {<element>}</code> <code>\prg_quicksort_compare:nnTF {<element₁>} {<element₂>}</code>
--	--

The two functions the programmer must define before calling `\prg_quicksort:n`. As an example we could define

```
\cs_set_nopar:Npn\prg_quicksort_function:n #1{{#1}}
\cs_set_nopar:Npn\prg_quicksort_compare:nnTF #1#2#3#4 {\int_compare:nNnTF{#1}>{#2}}
```

Then the function call

```
\prg_quicksort:n {876234520}
```

would return `{0}{2}{2}{3}{4}{5}{6}{7}{8}`. An alternative example where one sorts a list of words, `\prg_quicksort_compare:nnTF` could be defined as

```
\cs_set_nopar:Npn\prg_quicksort_compare:nnTF #1#2 {
  \int_compare:nNnTF{\tl_compare:nn{#1}{#2}}>\c_zero }
```


30.1 Variable type and scope

`\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.)

30.2 Mapping to variables

`\prg_new_map_functions:Nn` `\prg_new_map_functions:Nn <token> {\<name>}`

Creates a family of mapping functions which can be applied to a token list, dividing the list up at each occurrence of the `<token>`. The functions defined will be

- `\<name>_map_function:NN`
- `\<name>_map_function:nN`
- `\<name>_map_inline:Nn`
- `\<name>_map_inline:nn`
- `\<name>_map_break:`

Of these, the `inline` functions are not expandable but the other functions can be used in expansion contexts. The use of each function is best illustrated by the `\clist_map...` family defined by L^AT_EX3 itself for mapping to comma-separated lists. An error will be raised if the `<name>` has already been used to generate a family of mapping functions. All of the definitions are created globally.

`\prg_set_map_functions:Nn` `\prg_set_map_functions:Nn <token> {\<name>}`

Creates a family of mapping functions which can be applied to a token list, dividing the list up at each occurrence of the `<token>`. The functions defined will be

- `\<name>_map_function:NN`
- `\<name>_map_function:nN`
- `\<name>_map_inline:Nn`

- `\<name>_map_inline:nn`
- `\<name>_map_break:`

Of these, the `inline` functions are not expandable but the other functions can be used in expansion contexts. The use of each function is best illustrated by the `\clist_map_...` family defined by L^AT_EX3 itself for mapping to comma-separated lists. Any existing definitions for the `<name>` will be overwritten. All of the definitions are created globally.

Part VI

The l3quark package

“Quarks”

A special type of constants in L^AT_EX3 are ‘quarks’. These 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 is weird functions (for example as the stop token (i.e., `\q_stop`). They also permit the following ingenious trick: when you pick up a token in a temporary, and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary to the quark using `\if_meaning:w`. 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.

By convention all constants of type quark start out with `\q_`.

The documentation needs some updating.

31 Functions

<code>\quark_new:N</code>	<code>\quark_new:N <quark></code>
---------------------------	---

Defines `<quark>` to be a new constant of type `quark`.

<code>\quark_if_no_value_p:n *</code>	
<code>\quark_if_no_value:nTF *</code>	<code>\quark_if_no_value:nTF {<token list>} {<true code>} {<>false</code>
<code>\quark_if_no_value_p:N *</code>	<code>code>}</code>
<code>\quark_if_no_value:NTF *</code>	<code>\quark_if_no_value:NTF <tl var.> {<true code>} {<>false code>}</code>

This tests whether or not `<token list>` contains only the quark `\q_no_value`.

If $\langle token\ list \rangle$ to be tested is stored in a token list variable use `\quark_if_no_value:NTF`, or `\quark_if_no_value:NF` or check the value directly with `\if_meaning:w`. All those cases are faster then `\quark_if_no_value:nTF` so should be preferred.⁵

T_EXhackers note: But be aware of the fact that `\if_meaning:w` can result in an overflow of T_EX’s parameter stack since it leaves the corresponding `\fi:` on the input until the whole replacement text is processed. It is therefore better in recursions to use `\quark_if_no_value:NTF` as it will remove the conditional prior to processing the T or F case and so allows tail-recursion.

<code>\quark_if_nil_p:N *</code> <code>\quark_if_nil:N\underline{TF} *</code>	<code>\quark_if_nil:N\underline{TF} <token> {\<true code>} {\<false code>}</code>
---	--

This tests whether or not $\langle token \rangle$ is equal to the quark `\q_nil`.

This is a useful test for recursive loops which typically has `\q_nil` as an end marker.

<code>\quark_if_nil_p:n *</code> <code>\quark_if_nil:n\underline{TF} *</code> <code>\quark_if_nil_p:V *</code> <code>\quark_if_nil:V\underline{TF} *</code> <code>\quark_if_nil_p:o *</code> <code>\quark_if_nil:o\underline{TF} *</code>	<code>\quark_if_nil:N\underline{TF} {\<tokens>} {\<true code>} {\<false code>}</code>
---	--

This tests whether or not $\langle tokens \rangle$ is equal to the quark `\q_nil`.

This is a useful test for recursive loops which typically has `\q_nil` as an end marker.

32 Recursion

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

<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 *</code> <code>\quark_if_recursion_tail_stop:o *</code>	<code>\quark_if_recursion_tail_stop:n {\<list element>}</code> <code>\quark_if_recursion_tail_stop:N {\<list element>}</code>
--	--

⁵Clarify semantic of the “n” case ... i think it is not implement according to what we originally intended /FMi

This tests whether or not $\langle list\ element \rangle$ is equal to `\q_recursion_tail` and then exits, i.e., it gobbles the remainder of the list up to and including `\q_recursion_stop` which *must* be present.

If $\langle list\ element \rangle$ is not under your complete control it is advisable to use the `n`. If you wish to use the `N` form you *must* ensure it is really a single token such as if you have

```
\tl_set:Nn \l_tmpa_tl { \langle list element \rangle }
```

<code>\quark_if_recursion_tail_stop_do:Nn *</code>	<code>\quark_if_recursion_tail_stop_do:nn</code>
<code>\quark_if_recursion_tail_stop_do:nn *</code>	<code>{\langle list element \rangle} {\langle post action \rangle}</code>
<code>\quark_if_recursion_tail_stop_do:on *</code>	<code>\quark_if_recursion_tail_stop_do:Nn</code>
	<code>\langle list element \rangle {\langle post action \rangle}</code>

Same as `\quark_if_recursion_tail_stop:N` except here the second argument is executed after the recursion has been terminated.

33 Constants

`\q_no_value` The canonical ‘missing value quark’ that is returned by certain functions to denote that a requested value is not found in the data structure.

`\q_stop` This constant is used as a marker in parameter text. This allows a scanning function to find the end of some input string.

`\q_nil` This constant represent the nil pointer in pointer structures.

`\q_error` Delimits the end of the computation for purposes of error recovery.

`\q_mark` Used in parameter text when we need a scanning boundary that is distinct from `\q_stop`.

Part VII

The l3token package

A token of my appreciation...

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 \TeX , 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.

34 Character tokens

<code>\char_set_catcode:nn</code>	<code>\char_set_catcode:nn {<char number>} {<number>}</code>
<code>\char_set_catcode:w</code>	<code>\char_set_catcode:w <char> = <number></code>
<code>\char_value_catcode:n</code>	<code>\char_value_catcode:n {<char number>}</code>
<code>\char_value_catcode:w</code>	<code>\char_value_catcode:w {<char number>}</code>
<code>\char_show_value_catcode:n</code>	<code>\char_show_value_catcode:n {<char number>}</code>
<code>\char_show_value_catcode:w</code>	<code>\char_show_value_catcode:w {<char number>}</code>

`\char_set_catcode:nn` sets the category code of a character, `\char_value_catcode:n` returns its value for use in integer tests and `\char_show_value_catcode:n` pausing the typesetting and prints the value on the terminal and in the log file. The `:w` form should be avoided. (Will: should we then just not mention it?)

`\char_set_catcode` is more usefully abstracted below.

\TeX hackers note: `\char_set_catcode:w` is the \TeX primitive `\catcode` renamed.

<code>\char_make_escape:n</code>	
<code>\char_make_begin_group:n</code>	
<code>\char_make_end_group:n</code>	
<code>\char_make_math_shift:n</code>	
<code>\char_make_alignment:n</code>	
<code>\char_make_end_line:n</code>	
<code>\char_make_parameter:n</code>	
<code>\char_make_math_superscript:n</code>	
<code>\char_make_math_subscript:n</code>	
<code>\char_make_ignore:n</code>	
<code>\char_make_space:n</code>	
<code>\char_make_letter:n</code>	
<code>\char_make_other:n</code>	
<code>\char_make_active:n</code>	
<code>\char_make_comment:n</code>	<code>\char_make_letter:n {⟨character number⟩}</code>
<code>\char_make_invalid:n</code>	<code>\char_make_letter:n {64}</code>
	<code>\char_make_letter:n {'\@}</code>

Sets the catcode of the character referred to by its *⟨character number⟩*.

<code>\char_make_escape:N</code>	
<code>\char_make_begin_group:N</code>	
<code>\char_make_end_group:N</code>	
<code>\char_make_math_shift:N</code>	
<code>\char_make_alignment:N</code>	
<code>\char_make_end_line:N</code>	
<code>\char_make_parameter:N</code>	
<code>\char_make_math_superscript:N</code>	
<code>\char_make_math_subscript:N</code>	
<code>\char_make_ignore:N</code>	
<code>\char_make_space:N</code>	
<code>\char_make_letter:N</code>	
<code>\char_make_other:N</code>	
<code>\char_make_active:N</code>	
<code>\char_make_comment:N</code>	<code>\char_make_letter:N {⟨character⟩}</code>
<code>\char_make_invalid:N</code>	<code>\char_make_letter:N @</code>
	<code>\char_make_letter:N \%</code>

Sets the catcode of the *⟨character⟩*, which may have to be escaped.

T_EXhackers note: `\char_make_other:N` is L^AT_EX 2_ε's `\@makeother`.

<pre> \char_set_lccode:nn \char_set_lccode:w \char_value_lccode:n \char_value_lccode:w \char_show_value_lccode:n \char_show_value_lccode:w </pre>	<pre> \char_set_lccode:nn {<char>} {<number>} \char_set_lccode:w <char> = <number> \char_value_lccode:n {<char>} \char_show_value_lccode:n {<char>} </pre>
---	--

Set the lower case representation of $\langle char \rangle$ for when $\langle char \rangle$ is being converted in $\backslash tl_to_lowercase:n$. As above, the $:w$ form is only for people who really, really know what they are doing.

T_EXhackers note: $\backslash char_set_lccode:w$ is the T_EX primitive $\backslash lccode$ renamed.

<pre> \char_set_uccode:nn \char_set_uccode:w \char_value_uccode:n \char_value_uccode:w \char_show_value_uccode:n \char_show_value_uccode:w </pre>	<pre> \char_set_uccode:nn {<char>} {<number>} \char_set_uccode:w <char> = <number> \char_value_uccode:n {<char>} \char_show_value_uccode:n {<char>} </pre>
---	--

Set the uppercase representation of $\langle char \rangle$ for when $\langle char \rangle$ is being converted in $\backslash tl_to_uppercase:n$. As above, the $:w$ form is only for people who really, really know what they are doing.

T_EXhackers note: $\backslash char_set_uccode:w$ is the T_EX primitive $\backslash uccode$ renamed.

<pre> \char_set_sfcode:nn \char_set_sfcode:w \char_value_sfcode:n \char_value_sfcode:w \char_show_value_sfcode:n \char_show_value_sfcode:w </pre>	<pre> \char_set_sfcode:nn {<char>} {<number>} \char_set_sfcode:w <char> = <number> \char_value_sfcode:n {<char>} \char_show_value_sfcode:n {<char>} </pre>
---	--

Set the space factor for $\langle char \rangle$.

T_EXhackers note: $\backslash char_set_sfcode:w$ is the T_EX primitive $\backslash sfcode$ renamed.

<code>\char_set_mathcode:nn</code>	
<code>\char_set_mathcode:w</code>	
<code>\char_gset_mathcode:nn</code>	
<code>\char_gset_mathcode:w</code>	
<code>\char_value_mathcode:n</code>	<code>\char_set_mathcode:nn {<char>} {<number>}</code>
<code>\char_value_mathcode:w</code>	<code>\char_set_mathcode:w <char> = <number></code>
<code>\char_show_value_mathcode:n</code>	<code>\char_value_mathcode:n {<char>}</code>
<code>\char_show_value_mathcode:w</code>	<code>\char_show_value_mathcode:n {<char>}</code>

Set the math code for $\langle char \rangle$.

TeXhackers note: `\char_set_mathcode:w` is the TeX primitive `\mathcode` renamed.

35 Generic tokens

<code>\token_new:Nn</code>	<code>\token_new:Nn <token₁> {<token₂>}</code>
----------------------------	--

Defines $\langle token_1 \rangle$ to globally be a snapshot of $\langle token_2 \rangle$. This will be an implicit representation of $\langle token_2 \rangle$.

<code>\c_group_begin_token</code>
<code>\c_group_end_token</code>
<code>\c_math_shift_token</code>
<code>\c_alignment_tab_token</code>
<code>\c_parameter_token</code>
<code>\c_math_superscript_token</code>
<code>\c_math_subscript_token</code>
<code>\c_space_token</code>
<code>\c_letter_token</code>
<code>\c_other_char_token</code>
<code>\c_active_char_token</code>

Some useful constants. They have category codes 1, 2, 3, 4, 6, 7, 8, 10, 11, 12, and 13 respectively. They are all implicit tokens.

<code>\token_if_group_begin_p:N *</code>	
<code>\token_if_group_begin:N\underline{TF} *</code>	<code>\token_if_group_begin:N\underline{TF} <token> {<true>} {<false>}</code>

Check if $\langle token \rangle$ is a begin group token.

<code>\token_if_group_end_p:N *</code>	
<code>\token_if_group_end:N\underline{TF} *</code>	<code>\token_if_group_end:N\underline{TF} <token> {<true>} {<false>}</code>

Check if $\langle token \rangle$ is an end group token.

<code>\token_if_math_shift_p:N *</code>
<code>\token_if_math_shift:NTF *</code>

`\token_if_math_shift:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is a math shift token.

<code>\token_if_alignment_tab_p:N *</code>
<code>\token_if_alignment_tab:NTF *</code>

`\token_if_alignment_tab:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is an alignment tab token.

<code>\token_if_parameter_p:N *</code>
<code>\token_if_parameter:NTF *</code>

`\token_if_parameter:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is a parameter token.

<code>\token_if_math_superscript_p:N *</code>
<code>\token_if_math_superscript:NTF *</code>

`\token_if_math_superscript:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is a math superscript token.

<code>\token_if_math_subscript_p:N *</code>
<code>\token_if_math_subscript:NTF *</code>

`\token_if_math_subscript:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is a math subscript token.

<code>\token_if_space_p:N *</code>
<code>\token_if_space:NTF *</code>

`\token_if_space:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is a space token.

<code>\token_if_letter_p:N *</code>
<code>\token_if_letter:NTF *</code>

`\token_if_letter:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is a letter token.

<code>\token_if_other_char_p:N *</code>
<code>\token_if_other_char:NTF *</code>

`\token_if_other_char:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is an other char token.

<code>\token_if_active_char_p:N *</code>
<code>\token_if_active_char:NTF *</code>

`\token_if_active_char:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is an active char token.

<code>\token_if_eq_meaning_p:NN *</code>
<code>\token_if_eq_meaning:NNTF *</code>

`\token_if_eq_meaning:NNTF $\langle token_1 \rangle$ $\langle token_2 \rangle$ {\true} {\false}`

Check if the meaning of two tokens are identical.

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

`\token_if_eq_catcode:NNTF $\langle token_1 \rangle$ $\langle token_2 \rangle$ {\true} {\false}`

Check if the category codes of two tokens are equal. If both tokens are control sequences the test will be true.

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

`\token_if_eq_catcode:NNTF $\langle token_1 \rangle$ $\langle token_2 \rangle$ {\true} {\false}`

Check if the character codes of two tokens are equal. If both tokens are control sequences the test will be true.

<code>\token_if_macro_p:N *</code>
<code>\token_if_macro:NTF *</code>

`\token_if_macro:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is a macro.

<code>\token_if_cs_p:N *</code>
<code>\token_if_cs:NTF *</code>

`\token_if_cs:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is a control sequence or not. This can be useful for situations where the next token in the input stream is being looked at and you want to determine what should be done to it.

<code>\token_if_expandable_p:N *</code>
<code>\token_if_expandable:NTF *</code>

`\token_if_expandable:NTF $\langle token \rangle$ {\true} {\false}`

Check if $\langle token \rangle$ is expandable or not. Note that $\langle token \rangle$ can very well be an active character.

The next set of functions here are for picking apart control sequences. Sometimes it is useful to know if a control sequence has arguments and if so, how many. Similarly its status with respect to `\long` or `\protected` is good to have. Finally it can be very useful to know if a control sequence is of a certain type: Is this $\langle toks \rangle$ register we’re trying to to something with really a $\langle toks \rangle$ register at all?

<code>\token_if_long_macro_p:N *</code> <code>\token_if_long_macro:NTF *</code>	<code>\token_if_long_macro:NTF $\langle token \rangle$ {\true} {\false}</code>
--	---

Check if $\langle token \rangle$ is a “long” macro.

<code>\token_if_protected_macro_p:N *</code> <code>\token_if_protected_macro:NTF *</code>	<code>\token_if_protected_macro:NTF $\langle token \rangle$ {\true} {\false}</code>
--	--

Check if $\langle token \rangle$ is a “protected” macro. This test does *not* return $\langle true \rangle$ if the macro is also “long”, see below.

<code>\token_if_protected_long_macro_p:N *</code> <code>\token_if_protected_long_macro:NTF *</code>	<code>\token_if_protected_long_macro:NTF $\langle token \rangle$ {\true} {\false}</code>
--	---

Check if $\langle token \rangle$ is a “protected long” macro.

<code>\token_if_chardef_p:N *</code> <code>\token_if_chardef:NTF *</code>	<code>\token_if_chardef:NTF $\langle token \rangle$ {\true} {\false}</code>
--	--

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

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

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

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

Check if $\langle token \rangle$ is defined to be an integer register.

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

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

$\backslash token_if_skip_register_p:N \star$ $\backslash token_if_skip_register:N\mathit{TF} \star$	$\backslash token_if_skip_register:N\mathit{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	--

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

$\backslash token_if_toks_register_p:N \star$ $\backslash token_if_toks_register:N\mathit{TF} \star$	$\backslash token_if_toks_register:N\mathit{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	--

Check if $\langle token \rangle$ is defined to be a toks register.

$\backslash token_get_prefix_spec:N \star$ $\backslash token_get_arg_spec:N \star$ $\backslash token_get_replacement_spec:N \star$	$\backslash token_get_arg_spec:N \langle token \rangle$
---	--

If token is a macro with definition $\backslash cs_set:Npn\backslash next \#1\#2\{x'\#1--\#2'y\}$, the `prefix` function will return the string $\backslash long$, the `arg` function returns the string $\#1\#2$ and the `replacement` function returns the string $x'\#1--\#2'y$. If $\langle token \rangle$ isn't a macro, these functions return the $\backslash scan_stop$: token.

If the `arg_spec` contains the string \rightarrow , then the `spec` function will produce incorrect results.

35.1 Useless code: because we can!

$\backslash token_if_primitive_p:N \star$ $\backslash token_if_primitive:N\mathit{TF} \star$	$\backslash token_if_primitive:N\mathit{TF} \langle token \rangle \{\langle true \rangle\} \{\langle false \rangle\}$
--	---

Check if $\langle token \rangle$ is a primitive. Probably not a very useful function.

36 Peeking ahead at the next token

$\backslash l_peek_token$ $\backslash g_peek_token$ $\backslash l_peek_search_token$

Some useful variables. Initially they are set to ?.

$\backslash peek_after:NN$ $\backslash peek_gafter:NN$	$\backslash peek_after:NN \langle function \rangle \langle token \rangle$
---	--

Assign $\langle token \rangle$ to $\backslash l_peek_token$ and then run $\langle function \rangle$ which should perform some

sort of test on this token. Leaves $\langle token \rangle$ in the input stream. $\backslash peek_gafter:NN$ does this globally to the token $\backslash g_peek_token$.

T_EXhackers note: This is the primitive $\backslash futurelet$ turned into a function.

$\backslash peek_meaning:N\overline{TF}$ $\backslash peek_meaning_ignore_spaces:N\overline{TF}$ $\backslash peek_meaning_remove:N\overline{TF}$ $\backslash peek_meaning_remove_ignore_spaces:N\overline{TF}$	$\backslash peek_meaning:N\overline{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	--

$\backslash peek_meaning:N\overline{TF}$ checks (by using $\backslash if_meaning:w$) if $\langle token \rangle$ equals the next token in the input stream and executes either $\langle true\ code \rangle$ or $\langle false\ code \rangle$ accordingly. $\backslash peek_meaning_remove:N\overline{TF}$ does the same but additionally removes the token if found. The $ignore_spaces$ versions skips blank spaces before making the decision.

T_EXhackers note: This is equivalent to L^AT_EX 2_ε's $\backslash @ifnextchar$.

$\backslash peek_charcode:N\overline{TF}$ $\backslash peek_charcode_ignore_spaces:N\overline{TF}$ $\backslash peek_charcode_remove:N\overline{TF}$ $\backslash peek_charcode_remove_ignore_spaces:N\overline{TF}$	$\backslash peek_charcode:N\overline{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	---

Same as for the $\backslash peek_meaning:N\overline{TF}$ functions above but these use $\backslash if_charcode:w$ to compare the tokens.

$\backslash peek_catcode:N\overline{TF}$ $\backslash peek_catcode_ignore_spaces:N\overline{TF}$ $\backslash peek_catcode_remove:N\overline{TF}$ $\backslash peek_catcode_remove_ignore_spaces:N\overline{TF}$	$\backslash peek_catcode:N\overline{TF} \langle token \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	--

Same as for the $\backslash peek_meaning:N\overline{TF}$ functions above but these use $\backslash if_catcode:w$ to compare the tokens.

$\backslash peek_token_generic:N\overline{N\overline{TF}}$ $\backslash peek_token_remove_generic:N\overline{N\overline{TF}}$	$\backslash peek_token_generic:N\overline{N\overline{TF}} \langle token \rangle \langle function \rangle \{ \langle true \rangle \} \{ \langle false \rangle \}$
--	--

$\backslash peek_token_generic:N\overline{N\overline{TF}}$ looks ahead and checks if the next token in the input stream is equal to $\langle token \rangle$. It uses $\langle function \rangle$ to make that decision. $\backslash peek_token_remove_generic:N\overline{N\overline{TF}}$

does the same thing but additionally removes $\langle token \rangle$ from the input stream if it is found. This also works if $\langle token \rangle$ is either `\c_group_begin_token` or `\c_group_end_token`.

<code>\peek_execute_branches_meaning:</code> <code>\peek_execute_branches_charcode:</code> <code>\peek_execute_branches_catcode:</code>	<code>\peek_execute_branches_meaning:</code>
---	--

These functions compare the token we are searching for with the token found (after optional ignoring of specific tokens). They come in the usual three versions when T_EX is comparing tokens: meaning, character code, and category code.

Part VIII

The l3int package

Integers/counters

37 Integer values

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* (`'int expr'`).

37.1 Integer expressions

<code>\int_eval:n *</code>	<code>\int_eval:n {$\langle integer expression \rangle$}</code>
----------------------------	--

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

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

and

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

both evaluate to -6 . The $\{\langle integer\ expression\rangle\}$ may contain the operators $+$, $-$, $*$ and $/$, along with parenthesis $($ and $)$. After two expansions, `\int_eval:n` yields a $\langle integer\ donation\rangle$ which is left in the input stream. This is *not* an $\langle internal\ integer\rangle$, and therefore requires suitable termination if used in a T_EX-style integer assignment.

<code>\int_abs:n *</code>	<code>\int_abs:n {\langle integer\ expression\rangle}</code>
---------------------------	--

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

<code>\int_div_round:nn *</code>	<code>\int_div_round:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}</code>
----------------------------------	--

Evaluates the two $\langle integer\ expressions\rangle$ as described earlier, then calculates the result of dividing the first value by the second, rounding any remainder. Note that division using $/$ is identical to this function. The result is left in the input stream as a $\langle integer\ denotation\rangle$ after two expansions.

<code>\int_div_truncate:nn *</code>	<code>\int_div_truncate:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}</code>
-------------------------------------	---

Evaluates the two $\langle integer\ expressions\rangle$ 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 a $\langle integer\ denotation\rangle$ after two expansions.

<code>\int_max:nn *</code>	<code>\int_max:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}</code>
<code>\int_min:nn *</code>	<code>\int_min:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}</code>

Evaluates the $\langle integer\ expressions\rangle$ as described for `\int_eval:n` and leaves either the larger or smaller value in the input stream as an $\langle integer\ denotation\rangle$ after two expansions.

<code>\int_mod:nn *</code>	<code>\int_mod:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}</code>
----------------------------	--

Evaluates the two $\langle integer\ expressions\rangle$ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is left in the input stream as an $\langle integer\ denotation\rangle$ after two expansions.

37.2 Integer variables

<code>\int_new:N</code>	<code>\int_new:N \langle integer\rangle</code>
<code>\int_new:c</code>	

Creates a new $\langle integer\rangle$ or raises an error if the name is already taken. The declaration is

global. The $\langle integer \rangle$ will initially be equal to 0.

<code>\int_set_eq:NN</code>
<code>\int_set_eq:cN</code>
<code>\int_set_eq:Nc</code>
<code>\int_set_eq:cc</code>

`\int_set_eq:NN $\langle integer1 \rangle$ $\langle integer2 \rangle$`

Sets the content of $\langle integer1 \rangle$ equal to that of $\langle integer2 \rangle$. This assignment is restricted to the current \TeX group level.

<code>\int_gset_eq:NN</code>
<code>\int_gset_eq:cN</code>
<code>\int_gset_eq:Nc</code>
<code>\int_gset_eq:cc</code>

`\int_gset_eq:NN $\langle integer1 \rangle$ $\langle integer2 \rangle$`

Sets the content of $\langle integer1 \rangle$ equal to that of $\langle integer2 \rangle$. This assignment is global and so is not limited by the current \TeX group level.

<code>\int_add:Nn</code>
<code>\int_add:cn</code>

`\int_add:Nn $\langle integer \rangle$ $\{ \langle integer expression \rangle \}$`

Adds the result of the $\langle integer expression \rangle$ to the current content of the $\langle integer \rangle$. This assignment is local.

<code>\int_gadd:Nn</code>
<code>\int_gadd:cn</code>

`\int_gadd:Nn $\langle integer \rangle$ $\{ \langle integer expression \rangle \}$`

Adds the result of the $\langle integer expression \rangle$ to the current content of the $\langle integer \rangle$. This assignment is global.

<code>\int_decr:N</code>
<code>\int_decr:c</code>

`\int_decr:N $\langle integer \rangle$`

Decreases the value stored in $\langle integer \rangle$ by 1 within the scope of the current \TeX group.

<code>\int_gdecr:N</code>
<code>\int_gdecr:c</code>

`\int_incr:N $\langle integer \rangle$`

Decreases the value stored in $\langle integer \rangle$ by 1 globally (*i.e.* not limited by the current group level).

<code>\int_incr:N</code>
<code>\int_incr:c</code>

`\int_incr:N $\langle integer \rangle$`

Increases the value stored in $\langle integer \rangle$ by 1 within the scope of the current \TeX group.

<code>\int_gincr:N</code>
<code>\int_gincr:c</code>

`\int_incr:N $\langle integer \rangle$`

Increases the value stored in $\langle integer \rangle$ by 1 globally (*i.e.* not limited by the current group level).

level).

<code>\int_set:Nn</code>
<code>\int_set:cn</code>

`\int_set:Nn` $\langle integer \rangle$ $\{ \langle integer expression \rangle \}$

Sets $\langle integer \rangle$ to the value of $\langle integer expression \rangle$, which must evaluate to an integer (as described for `\int_eval:n`). This assignment is restricted to the current TeX group.

<code>\int_gset:Nn</code>
<code>\int_gset:cn</code>

`\int_gset:Nn` $\langle integer \rangle$ $\{ \langle integer expression \rangle \}$

Sets $\langle integer \rangle$ to the value of $\langle integer expression \rangle$, which must evaluate to an integer (as described for `\int_eval:n`). This assignment is global and is not limited to the current TeX group level.

<code>\int_sub:Nn</code>
<code>\int_sub:cn</code>

`\int_sub:Nn` $\langle integer \rangle$ $\{ \langle integer expression \rangle \}$

Subtracts the result of the $\langle integer expression \rangle$ to the current content of the $\langle integer \rangle$. This assignment is local.

<code>\int_gsub:Nn</code>
<code>\int_gsub:cn</code>

`\int_gsub:Nn` $\langle integer \rangle$ $\{ \langle integer expression \rangle \}$

Subtracts the result of the $\langle integer expression \rangle$ to the current content of the $\langle integer \rangle$. This assignment is global.

<code>\int_zero:N</code>
<code>\int_zero:c</code>

`\int_zero:N` $\langle integer \rangle$

Sets $\langle integer \rangle$ to 0 within the scope of the current TeX group.

<code>\int_gzero:N</code>
<code>\int_gzero:c</code>

`\int_gzero:N` $\langle integer \rangle$

Sets $\langle integer \rangle$ to 0 globally, *i.e.* not restricted by the current TeX group level.

<code>\int_show:N</code>
<code>\int_show:c</code>

`\int_show:N` $\langle integer \rangle$

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

<code>\int_use:N *</code>
<code>\int_use:c *</code>

`\int_use:N` $\langle integer \rangle$

Recovers the content of a $\langle integer \rangle$ 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 $\langle integer \rangle$ is required (such as in the first and third arguments of `\int_compare:nNnTF`).

37.3 Comparing integer expressions

	<code>\int_compare_p:nNn</code>	<code>{⟨integer expression 1⟩} ⟨relation⟩ {⟨integer expression 2⟩}</code>
<code>\int_compare_p:nNn *</code>	<code>\int_compare:nNnTF</code>	<code>{⟨integer expression 1⟩} ⟨relation⟩ {⟨integer expression 2⟩}</code>
<code>\int_compare:nNnTF *</code>		<code>{⟨true code⟩} {⟨false code⟩}</code>

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

Equal	=
Greater than	>
Less than	<

The branching versions then leave either *⟨true code⟩* or *⟨false code⟩* in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

	<code>\int_compare_p:n</code>	<code>{ ⟨integer expression 1⟩ ⟨relation⟩ ⟨integer expression 2⟩ }</code>
<code>\int_compare_p:n *</code>	<code>\int_compare:nTF</code>	<code>{ ⟨integer expression 1⟩ ⟨relation⟩ ⟨integer expression 2⟩ }</code>
<code>\int_compare:nTF *</code>		<code>{⟨true code⟩} {⟨false code⟩}</code>

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

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

The branching versions then leave either *⟨true code⟩* or *⟨false code⟩* in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

<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>\int_if_odd:nTF *</code>	<code>{⟨true code⟩} {⟨false code⟩}</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. The branching versions then leave

either $\langle true\ code \rangle$ or $\langle false\ code \rangle$ in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

<code>\int_do_while:nNnn *</code>	<code>\int_do_while:nNnn</code> $\{ \langle intexpr_1 \rangle \} \langle relation \rangle \{ \langle intexpr_2 \rangle \} \{ \langle code \rangle \}$
-----------------------------------	--

Evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for `\int_compare:nNnTF`, and then places the $\langle code \rangle$ in the input stream if the $\langle relation \rangle$ is **true**. 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 **false**.

<code>\int_do_until:nNnn *</code>	<code>\int_do_until:nNnn</code> $\{ \langle intexpr_1 \rangle \} \langle relation \rangle \{ \langle intexpr_2 \rangle \} \{ \langle code \rangle \}$
-----------------------------------	--

Evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for `\int_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**.

<code>\int_until_do:nNnn *</code>	<code>\int_until_do:nNnn</code> $\{ \langle intexpr_1 \rangle \} \langle relation \rangle \{ \langle intexpr_2 \rangle \} \{ \langle code \rangle \}$
-----------------------------------	--

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for `\int_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>\int_while_do:nNnn *</code>	<code>\int_while_do:nNnn</code> $\{ \langle intexpr_2 \rangle \} \{ \langle code \rangle \}$	$\{ \langle intexpr_1 \rangle \} \langle relation \rangle$
-----------------------------------	---	--

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for `\int_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>\int_do_while:nn *</code>	<code>\int_do_while:nn</code> $\{ \langle intexpr_1 \rangle \} \langle relation \rangle \{ \langle intexpr_2 \rangle \} \{ \langle code \rangle \}$
---------------------------------	--

Evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for `\int_compare:nTF`, and then places the $\langle code \rangle$ in the input stream if the $\langle relation \rangle$ is **true**. 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 **false**.

<code>\int_do_until:nn *</code>	<code>\int_do_until:nn</code> $\{ \langle intexpr_1 \rangle \} \langle relation \rangle \{ \langle intexpr_2 \rangle \} \{ \langle code \rangle \}$
---------------------------------	--

Evaluates the relationship between the two $\langle integer\ expressions \rangle$ as described for `\int_compare:nTF`, 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**.

\int_until_do:nn *	$\backslash\mathrm{int_until_do:nn}$ $\{ \langle\mathit{intexpr1}\rangle \langle\mathit{relation}\rangle \langle\mathit{intexpr2}\rangle \} \{ \langle\mathit{code}\rangle \}$
--------------------	---

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle\mathit{integer expressions}\rangle$ as described for $\backslash\mathrm{int_compare:nTF}$. 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 $\langle\mathit{relation}\rangle$ is **true**.

\int_while_do:nn *	$\backslash\mathrm{int_while_do:nn}$ $\{ \langle\mathit{intexpr1}\rangle \langle\mathit{relation}\rangle \langle\mathit{intexpr2}\rangle \}$ $\{ \langle\mathit{code}\rangle \}$
--------------------	--

Places the $\langle code \rangle$ in the input stream for T_EX to process, and then evaluates the relationship between the two $\langle\mathit{integer expressions}\rangle$ as described for $\backslash\mathrm{int_compare:nTF}$. 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 $\langle\mathit{relation}\rangle$ is **false**.

37.4 Formatting integers

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

\int_to_arabic:n *	$\backslash\mathrm{int_to_arabic:n}$ $\{ \langle\mathit{integer expression}\rangle \}$
--------------------	--

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

\int_to_alph:n *	$\backslash\mathrm{int_to_alph:n}$ $\{ \langle\mathit{integer expression}\rangle \}$
\int_to_Alph:n *	

Evaluates the $\langle\mathit{integer expression}\rangle$ and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order. Thus

$\backslash\mathrm{int_to_alph:n}$ { 1 }

places **a** in the input stream,

$\backslash\mathrm{int_to_alph:n}$ { 26 }

is represented as **z** and

$\backslash\mathrm{int_to_alph:n}$ { 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_binary:n *` `\int_to_binary:n {⟨integer expression⟩}`

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

`\int_to_hexadecimal:n *` `\int_to_hexadecimal:n {⟨integer expression⟩}`

Calculates the value of the *⟨integer expression⟩* and places the hexadecimal (base 16) representation of the result in the input stream. Upper case letters are used for digits beyond 9.

`\int_to_octal:n *` `\int_to_octal:n {⟨integer expression⟩}`

Calculates the value of the *⟨integer expression⟩* and places the octal (base 8) representation of the result in the input stream.

`\int_to_roman:n *`
`\int_to_Roman:n *` `\int_to_roman:n {⟨integer expression⟩}`

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

`\int_to_symbol:n *` `\int_to_symbol:n {⟨integer expression⟩}`

Calculates the value of the *⟨integer expression⟩* and places the symbol representation of the result in the input stream. The list of symbols used is equivalent to L^AT_EX 2_ε’s `\@fnsymbol` set.

37.5 Converting from other formats

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

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

`\int_from_binary:n *` `\int_from_binary:n {⟨binary number⟩}`

Converts the $\langle binary\ number \rangle$ into the integer (base 10) representation and leaves this in the input stream.

```
\int_from_hexadecimal:n * \int_from_binary:n {\hexadecimal number}
```

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.

```
\int_from_octal:n * \int_from_octal:n {\octal number}
```

Converts the $\langle octal\ number \rangle$ into the integer (base 10) representation and leaves this in the input stream.

```
\int_from_roman:n * \int_from_roman:n {\roman numeral}
```

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 .

37.6 Low-level conversion functions

As well as the higher-level functions already documented, there are a series of lower-level functions which can be used to carry out generic conversions. These are used to create the higher-level versions documented above.

```
\int_convert_from_base_ten:nn * \int_convert_from_base_ten:nn {\integer expression}
{\base}
```

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 (with normal category codes). The maximum $\langle base \rangle$ value is 36.

```
\int_convert_to_base_ten:nn * \int_convert_to_base_ten:nn {\number}
{\base}
```

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.

```
\int_convert_to_symbols:nnn
{\integer expression} {\total symbols}
\int_convert_to_symbols:nnn * {\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_sybols:nnn {#1} { 26 }
  {
    { 1 } { a }
    { 2 } { b }
    { 3 } { c }
    { 4 } { d }
    { 5 } { e }
    { 6 } { f }
    { 7 } { g }
    { 8 } { h }
    { 9 } { i }
    { 10 } { j }
    { 11 } { k }
    { 12 } { l }
    { 13 } { m }
    { 14 } { n }
    { 15 } { o }
    { 16 } { p }
    { 17 } { q }
    { 18 } { r }
    { 19 } { s }
    { 20 } { t }
    { 21 } { u }
    { 22 } { v }
    { 23 } { w }
    { 24 } { x }
    { 25 } { y }
    { 26 } { z }
  }
}
```

38 Variables and constants

<code>\l_tmpa_int</code>
<code>\l_tmpb_int</code>
<code>\l_tmpc_int</code>
<code>\g_tmpa_int</code>
<code>\g_tmpb_int</code>

Scratch register for immediate use. They are not used by conditionals or predicate functions.

<code>\int_const:Nn</code>
<code>\int_const:cn</code>

`\int_const:Nn` $\langle integer \rangle$ $\{ \langle integer expression \rangle \}$

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 $\langle integer expression \rangle$.

`\c_max_int` The maximum value that can be stored as an integer.

```
\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_hundred_one
\c_twohundred_fifty_five
\c_twohundred_fifty_six
\c_thousand
\c_ten_thousand
\c_ten_thousand_one
\c_ten_thousand_two
\c_ten_thousand_three
\c_ten_thousand_four
\c_twenty_thousand
```

Integer values used with primitive tests and assignments:
self-terminating nature makes these more convenient and faster than literal numbers.

`\c_max_register_int` Maximum number of registers.

38.1 Internal functions

`\int_to_roman:w *` `\int_to_roman:w` $\langle integer \rangle$ $\langle space \rangle$ or $\langle non-expandable token \rangle$

Converts $\langle integer \rangle$ to it lowercase roman representation. Note that it produces a string of letters with catcode 12.

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

<code>\int_roman_lcuc_mapping:Nnn</code>	<code>\int_roman_lcuc_mapping:Nnn <roman_char> {\<licr>}</code>
<code>\int_to_roman_lcuc:NN</code>	<code>{\<LICR>}</code>
	<code>\int_to_roman_lcuc:NN <roman_char> <char></code>

`\int_roman_lcuc_mapping:Nnn` specifies how the roman numeral `<roman_char>` (i, v, x, l, c, d, or m) should be interpreted when converting the number. `<licr>` is the lower case and `<LICR>` is the uppercase mapping. `\int_to_roman_lcuc:NN` is a recursive function converting the roman numerals.

<code>\int_convert_number_with_rule:nnN</code>	
<code>\int_symbol_math_conversion_rule:n</code>	<code>\int_convert_number_with_rule:nnN {\<int₁>} {\<int₂>}</code>
<code>\int_symbol_text_conversion_rule:n</code>	<code><function></code>

`\int_convert_number_with_rule:nnN` converts `<int1>` into letters, symbols, whatever as defined by `<function>`. `<int2>` denotes the base number for the conversion.

<code>\if_num:w</code>	<code>*</code>	<code>\if_num:w <number₁> <rel> <number₂> <true> \else: <false></code>
<code>\if_int_compare:w</code>	<code>*</code>	<code>\fi:</code>

Compare two integers using `<rel>`, which must be one of =, < or > with category code 12. The `\else:` branch is optional.

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

<code>\if_case:w</code>	<code>*</code>	<code>\if_case:w <number> <case₀> \or: <case₁> \or: ... \else:</code>
<code>\or:</code>	<code>*</code>	<code><default> \fi:</code>

Selects a case to execute based on the value of `<number>`. The first case (`<case0>`) is executed if `<number>` is 0, the second (`<case1>`) if the `<number>` is 1, etc. The `<number>` may be a literal, a constant or an integer expression (e.g. using `\int_eval:n`).

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

<code>\int_value:w</code>	<code>*</code>	<code>\int_value:w <integer></code>
		<code>\int_value:w <tokens> <optional space></code>

Expands `<tokens>` until an `<integer>` 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>*</code>
<code>\int_eval_end:</code>	<code>*</code>

`\int_eval:w` $\langle int\ expr \rangle$ `\int_eval_end:`

Evaluates $\langle integer\ expression \rangle$ as described for `\int_eval:n`. The evaluation stops when an unexpandable token with category code other than 12 is read or when `\int_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `\int_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

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

<code>\if_int_odd:w</code>	<code>*</code>
----------------------------	----------------

`\if_int_odd:w` $\langle tokens \rangle$ $\langle true \rangle$ `\else:` $\langle false \rangle$ `\fi:`

`\if_int_odd:w` $\langle number \rangle$ $\langle true \rangle$ `\else:` $\langle false \rangle$ `\fi:`

Expands $\langle tokens \rangle$ until a non-numeric tokens is found, and tests whether the resulting $\langle number \rangle$ is odd. If so, $\langle true\ code \rangle$ is executed. The `\else:` branch is optional.

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

Part IX

The l3skip package

Dimension and skip registers

L^AT_EX3 knows about two types of length registers for internal use: rubber lengths (**skips**) and rigid lengths (**dims**).

39 Skip registers

39.1 Functions

<code>\skip_new:N</code>
<code>\skip_new:c</code>

`\skip_new:N` $\langle skip \rangle$

Defines $\langle skip \rangle$ to be a new variable of type **skip**.

TeXhackers note: `\skip_new:N` is the equivalent to plain TeX's `\newskip`.

<code>\skip_zero:N</code>
<code>\skip_zero:c</code>
<code>\skip_gzero:N</code>
<code>\skip_gzero:c</code>

`\skip_zero:N <skip>`

Locally or globally reset $\langle skip \rangle$ to zero. For global variables the global versions should be used.

<code>\skip_set:Nn</code>
<code>\skip_set:cn</code>
<code>\skip_gset:Nn</code>
<code>\skip_gset:cn</code>

`\skip_set:Nn <skip> {<skip value>}`

These functions will set the $\langle skip \rangle$ register to the $\langle length \rangle$ value.

<code>\skip_add:Nn</code>
<code>\skip_add:cn</code>
<code>\skip_gadd:Nn</code>
<code>\skip_gadd:cn</code>

`\skip_add:Nn <skip> {<length>}`

These functions will add to the $\langle skip \rangle$ register the value $\langle length \rangle$. If the second argument is a $\langle skip \rangle$ register too, the surrounding braces can be left out.

<code>\skip_sub:Nn</code>
<code>\skip_gsub:Nn</code>

`\skip_gsub:Nn <skip> {<length>}`

These functions will subtract from the $\langle skip \rangle$ register the value $\langle length \rangle$. If the second argument is a $\langle skip \rangle$ register too, the surrounding braces can be left out.

<code>\skip_use:N</code>
<code>\skip_use:c</code>

`\skip_use:N <skip>`

This function returns the length value kept in $\langle skip \rangle$ in a way suitable for further processing.

T_EXhackers note: The function `\skip_use:N` could be implemented directly as the T_EX primitive `\tex_the:D` which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explanatory.

<code>\skip_show:N</code>
<code>\skip_show:c</code>

`\skip_show:N <skip>`

This function pauses the compilation and displays the length value kept in $\langle skip \rangle$ in the console output and log file.

T_EXhackers note: The function `\skip_show:N` could be implemented directly as the T_EX primitive `\tex_showthe:D` which is also responsible to produce the values for other internal

quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explanatory.

<code>\skip_horizontal:N</code>
<code>\skip_horizontal:c</code>
<code>\skip_horizontal:n</code>
<code>\skip_vertical:N</code>
<code>\skip_vertical:c</code>
<code>\skip_vertical:n</code>

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

The `hor` functions insert `<skip>` or `<length>` with the \TeX primitive `\hskip`. The `vertical` variants do the same with `\vskip`. The `n` versions evaluate `<length>` with `\skip_eval:n`.

<code>\skip_if_infinite_glue_p:n</code>
<code>\skip_if_infinite_glue:nTF</code>

`\skip_if_infinite_glue:nTF {<skip>} {<true>} {<false>}`

Checks if `<skip>` contains infinite stretch or shrink components and executes either `<true>` or `<false>`. Also works on input like `3pt plus .5in`.

<code>\skip_split_finite_else_action:nnNN</code>
--

`\skip_split_finite_else_action:nnNN {<skip>} {<action>}`
`<dimen1> <dimen2>`

Checks if `<skip>` 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 zero and execute `#2` which is usually an error or warning message of some sort.

<code>\skip_eval:n</code>

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

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

39.2 Formatting a skip register value

39.3 Variable and constants

<code>\c_max_skip</code>	Constant that denotes the maximum value which can be stored in a $\langle skip \rangle$ register.
--------------------------	---

<code>\c_zero_skip</code>	Constants denoting a zero skip.
---------------------------	---------------------------------

<code>\l_tmpa_skip</code> <code>\l_tmpb_skip</code> <code>\l_tmpc_skip</code> <code>\g_tmpa_skip</code> <code>\g_tmpb_skip</code>	Scratch register for immediate use.
---	-------------------------------------

40 Dim registers

40.1 Functions

<code>\dim_new:N</code> <code>\dim_new:c</code>	<code>\dim_new:N</code> $\langle dim \rangle$
--	---

Defines $\langle dim \rangle$ to be a new variable of type `dim`.

T_EXhackers note: `\dim_new:N` is the equivalent to plain T_EX's `\newdimen`.

<code>\dim_zero:N</code> <code>\dim_zero:c</code> <code>\dim_gzero:N</code> <code>\dim_gzero:c</code>	<code>\dim_zero:N</code> $\langle dim \rangle$
--	--

Locally or globally reset $\langle dim \rangle$ to zero. For global variables the global versions should be

used.

<code>\dim_set:Nn</code>
<code>\dim_set:Nc</code>
<code>\dim_set:cn</code>
<code>\dim_gset:Nn</code>
<code>\dim_gset:Nc</code>
<code>\dim_gset:cn</code>
<code>\dim_gset:cc</code>

`\dim_set:Nn ⟨dim⟩ {⟨dim value⟩}`

These functions will set the $\langle dim \rangle$ register to the $\langle dim value \rangle$ value.

<code>\dim_set_max:Nn</code>
<code>\dim_set_max:cn</code>

`\dim_set_max:Nn ⟨dimension⟩ {⟨dimension expression⟩}`

Compares the current value of the $\langle dimension \rangle$ with that of the $\langle dimension expression \rangle$, and sets the $\langle dimension \rangle$ to the larger of these two value. This assignment is local to the current \TeX group.

<code>\dim_gset_max:Nn</code>
<code>\dim_gset_max:cn</code>

`\dim_gset_max:Nn ⟨dimension⟩ {⟨dimension expression⟩}`

Compares the current value of the $\langle dimension \rangle$ with that of the $\langle dimension expression \rangle$, and sets the $\langle dimension \rangle$ to the larger of these two value. This assignment is global.

<code>\dim_set_min:Nn</code>
<code>\dim_set_min:cn</code>

`\dim_set_min:Nn ⟨dimension⟩ {⟨dimension expression⟩}`

Compares the current value of the $\langle dimension \rangle$ with that of the $\langle dimension expression \rangle$, and sets the $\langle dimension \rangle$ to the smaller of these two value. This assignment is local to the current \TeX group.

<code>\dim_gset_min:Nn</code>
<code>\dim_gset_min:cn</code>

`\dim_gset_min:Nn ⟨dimension⟩ {⟨dimension expression⟩}`

Compares the current value of the $\langle dimension \rangle$ with that of the $\langle dimension expression \rangle$, and sets the $\langle dimension \rangle$ to the smaller of these two value. This assignment is global.

<code>\dim_add:Nn</code>
<code>\dim_add:Nc</code>
<code>\dim_add:cn</code>
<code>\dim_gadd:Nn</code>
<code>\dim_gadd:cn</code>

`\dim_add:Nn ⟨dim⟩ {⟨length⟩}`

These functions will add to the $\langle dim \rangle$ register the value $\langle length \rangle$. If the second argument

is a $\langle dim \rangle$ register too, the surrounding braces can be left out.

<code>\dim_sub:Nn</code>
<code>\dim_sub:Nc</code>
<code>\dim_sub:cn</code>
<code>\dim_gsub:Nn</code>
<code>\dim_gsub:cn</code>

`\dim_gsub:Nn $\langle dim \rangle$ { $\langle length \rangle$ }`

These functions will subtract from the $\langle dim \rangle$ register the value $\langle length \rangle$. If the second argument is a $\langle dim \rangle$ register too, the surrounding braces can be left out.

<code>\dim_use:N</code>
<code>\dim_use:c</code>

`\dim_use:N $\langle dim \rangle$`

This function returns the length value kept in $\langle dim \rangle$ in a way suitable for further processing.

T_EXhackers note: The function `\dim_use:N` could be implemented directly as the T_EX primitive `\tex_the:D` which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explanatory.

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

`\dim_show:N $\langle dim \rangle$`

This function pauses the compilation and displays the length value kept in $\langle skip \rangle$ in the console output and log file.

T_EXhackers note: The function `\dim_show:N` could be implemented directly as the T_EX primitive `\tex_showthe:D` which is also responsible to produce the values for other internal quantities. We have chosen to use individual functions for counters, dimensions etc. to allow checks and to make the code more self-explanatory.

<code>\dim_eval:n</code>

`\dim_eval:n { $\langle dimension expression \rangle$ }`

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

<code>\if_dim:w</code>

`\if_dim:w $\langle dimen_1 \rangle$ $\langle rel \rangle$ $\langle dimen_2 \rangle$ $\langle true \rangle$ \else: $\langle false \rangle$ \fi:`

Compare two dimensions. It is recommended to use `\dim_eval:w` to correctly evaluate and terminate these numbers. $\langle rel \rangle$ is one of `<`, `=` or `>` with catcode 12.

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

<code>\dim_compare_p:n *</code>	<code>\dim_compare_p:n {⟨⟨dim expr. 1⟩ ⟨rel⟩ ⟨dim expr. 2⟩⟩}</code>
<code>\dim_compare:nTF *</code>	<code>\dim_compare:nTF {⟨⟨dim expr. 1⟩ ⟨rel⟩ ⟨dim expr. 2⟩⟩}</code> <code>⟨true code⟩ ⟨false code⟩</code>

Evaluates $\langle dim\ expr. 1 \rangle$ and $\langle dim\ expr. 2 \rangle$ and then carries out a comparison of the resulting lengths using C-like operators:

Less than	<code><</code>	Less than or equal	<code><=</code>
Greater than	<code>></code>	Greater than or equal	<code>>=</code>
Equal	<code>==</code> or <code>=</code>	Not equal	<code>!=</code>

Based on the result of the comparison either the $\langle true\ code \rangle$ or $\langle false\ code \rangle$ is executed. Both dimension expressions are evaluated fully in the process. Note the syntax, which allows natural input in the style of

```
\dim_compare_p:n {2.54cm != \l_tmpb_int}
```

A single equals sign is available as comparator (in addition to those familiar to C users) as standard T_EX practice is to compare values using `=`.

<code>\dim_compare:nNnTF *</code>	<code>\dim_compare:nNnTF {⟨dim expr⟩} ⟨rel⟩ {⟨dim expr⟩}</code>
<code>\dim_compare_p:nNnTF *</code>	<code>{⟨true⟩} {⟨false⟩}</code>

These functions test two dimension expressions against each other. They are both evaluated by `\dim_eval:n`. Note that if both expressions are normal dimension variables as in

```
\dim_compare:nNnTF \l_temp_dim < \c_zero_skip {negative}{non-negative}
```

you can safely omit the braces.

These functions are faster than the `n` variants described above but do not support an extended set of relational operators.

T_EXhackers note: This is the T_EX primitive `\ifdim` turned into a function.

<code>\dim_while_do:nNnn</code>	<code>\dim_while_do:nNnn</code>	<code>⟨dim expr⟩ ⟨rel⟩ ⟨dim expr⟩ ⟨code⟩</code>
<code>\dim_until_do:nNnn</code>		
<code>\dim_do_while:nNnn</code>		
<code>\dim_do_until:nNnn</code>		

`\dim_while_do:nNnn` tests the dimension expressions and if true performs $\langle code \rangle$ repeatedly while the test remains true. `\dim_do_while:nNnn` is similar but executes the body first and then performs the check, thus ensuring that the body is executed at least once. The ‘until’ versions are similar but continue the loop as long as the test is false.

40.2 Variable and constants

`\c_max_dim` Constant that denotes the maximum value which can be stored in a $\langle dim \rangle$ register.

`\c_zero_dim` Set of constants denoting useful values.

<code>\l_tmpa_dim</code> <code>\l_tmpb_dim</code> <code>\l_tmpc_dim</code> <code>\l_tmpd_dim</code> <code>\g_tmpa_dim</code> <code>\g_tmpb_dim</code>	Scratch register for immediate use.
--	-------------------------------------

41 Muskips

`\muskip_new:N` $\backslash\text{muskip_new:N}$ $\langle muskip \rangle$

T_EXhackers note: Defines $\langle muskip \rangle$ to be a new variable of type `muskip`. `\muskip_new:N` is the equivalent to plain T_EX's `\newmuskip`.

<code>\muskip_set:Nn</code> <code>\muskip_gset:Nn</code>	$\backslash\text{muskip_set:Nn}$ $\langle muskip \rangle$ $\{\langle muskip\ value \rangle\}$ These functions will set the $\langle muskip \rangle$ register to the $\langle length \rangle$ value.
---	---

<code>\muskip_add:Nn</code> <code>\muskip_gadd:Nn</code>	$\backslash\text{muskip_add:Nn}$ $\langle muskip \rangle$ $\{\langle length \rangle\}$ These functions will add to the $\langle muskip \rangle$ register the value $\langle length \rangle$. If the second argument is a $\langle muskip \rangle$ register too, the surrounding braces can be left out.
---	--

<code>\muskip_sub:Nn</code> <code>\muskip_gsub:Nn</code>	$\backslash\text{muskip_gsub:Nn}$ $\langle muskip \rangle$ $\{\langle length \rangle\}$ These functions will subtract from the $\langle muskip \rangle$ register the value $\langle length \rangle$. If the second argument is a $\langle muskip \rangle$ register too, the surrounding braces can be left out.
---	--

`\muskip_use:N` $\backslash\text{muskip_use:N}$ $\langle muskip \rangle$

This function returns the length value kept in $\langle muskip \rangle$ in a way suitable for further processing.

T_EXhackers note: See note for `\dim_use:N`.

<code>\muskip_show:N</code>

`\muskip_show:N` $\langle muskip \rangle$

This function pauses the compilation and displays the length value kept in $\langle muskip \rangle$ in the console output and log file.

Part X

The l3tl package

Token Lists

L^AT_EX3 stores token lists in variables also called ‘token lists’. Variables of this type get the suffix `tl` and functions of this type have the prefix `tl`. To use a token list variable you simply call the corresponding variable.

Often you find yourself with not a token list variable but an arbitrary token list which has to undergo certain tests. We will *also* prefix these functions with `tl`. While token list variables are always single tokens, token lists are always surrounded by braces.

42 Functions

<code>\tl_new:N</code>
<code>\tl_new:c</code>
<code>\tl_new:Nn</code>
<code>\tl_new:cn</code>
<code>\tl_new:Nx</code>

`\tl_new:Nn` $\langle tl var. \rangle$ $\{ \langle initial token list \rangle \}$

Defines $\langle tl var. \rangle$ globally to be a new variable to store a token list. $\langle initial token list \rangle$ is the initial value of $\langle tl var. \rangle$. This makes it possible to assign values to a constant token list variable.

The form `\tl_new:N` initializes the token list variable with an empty value.

<code>\tl_const:Nn</code>

`\tl_const:Nn` $\langle tl var. \rangle$ $\{ \langle token list \rangle \}$

Defines $\langle tl var. \rangle$ as a global constant expanding to $\langle token list \rangle$. The name of the constant must be free when the constant is created.

<code>\tl_use:N</code>
<code>\tl_use:c</code>

`\tl_use:N` $\langle tl var. \rangle$

Function that inserts the $\langle tl var. \rangle$ into the processing stream. Instead of `\tl_use:N`

simply placing the $\langle tl\ var.\rangle$ into the input stream is also supported. `\tl_use:c` will complain if the $\langle tl\ var.\rangle$ hasn't been declared previously!

<code>\tl_show:N</code>	
<code>\tl_show:c</code>	<code>\tl_show:N $\langle tl\ var.\rangle$</code>
<code>\tl_show:n</code>	<code>\tl_show:n {$\langle token\ list\rangle$}</code>

Function that pauses the compilation and displays the $\langle tl\ var.\rangle$ or $\langle token\ list\rangle$ on the console output and in the log file.

<code>\tl_set:Nn</code>	
<code>\tl_set:Nc</code>	
<code>\tl_set:NV</code>	
<code>\tl_set:No</code>	
<code>\tl_set:Nv</code>	
<code>\tl_set:Nf</code>	
<code>\tl_set:Nx</code>	
<code>\tl_set:cn</code>	
<code>\tl_set:co</code>	
<code>\tl_set:cV</code>	
<code>\tl_set:cx</code>	
<code>\tl_gset:Nn</code>	
<code>\tl_gset:Nc</code>	
<code>\tl_gset:No</code>	
<code>\tl_gset:NV</code>	
<code>\tl_gset:Nv</code>	
<code>\tl_gset:Nx</code>	
<code>\tl_gset:cn</code>	
<code>\tl_gset:cx</code>	

`\tl_set:Nn $\langle tl\ var.\rangle$ { $\langle token\ list\rangle$ }`

Defines $\langle tl\ var.\rangle$ to hold the token list $\langle token\ list\rangle$. Global variants of this command assign the value globally the other variants expand the $\langle token\ list\rangle$ up to a certain level before the assignment or interpret the $\langle token\ list\rangle$ as a character list and form a control sequence out of it.

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

`\tl_clear:N $\langle tl\ var.\rangle$`

The $\langle tl\ var.\rangle$ is locally or globally cleared. The `c` variants will generate a control sequence name which is then interpreted as $\langle tl\ var.\rangle$ before clearing.

<code>\tl_clear_new:N</code>	
<code>\tl_clear_new:c</code>	
<code>\tl_gclear_new:N</code>	
<code>\tl_gclear_new:c</code>	

`\tl_clear_new:N $\langle tl\ var.\rangle$`

These functions check if $\langle tl\ var.\rangle$ exists. If it does it will be cleared; if it doesn't it will be allocated.

<code>\tl_put_left:Nn</code>
<code>\tl_put_left:NV</code>
<code>\tl_put_left:No</code>
<code>\tl_put_left:Nx</code>
<code>\tl_put_left:cn</code>
<code>\tl_put_left:cV</code>
<code>\tl_put_left:co</code>

`\tl_put_left:Nn $\langle tl\ var.\rangle$ {\mathit{token list}}`

These functions will append $\langle token\ list\rangle$ to the left of $\langle tl\ var.\rangle$. $\langle token\ list\rangle$ might be subject to expansion before assignment.

<code>\tl_put_right:Nn</code>
<code>\tl_put_right:NV</code>
<code>\tl_put_right:No</code>
<code>\tl_put_right:Nx</code>
<code>\tl_put_right:cn</code>
<code>\tl_put_right:cV</code>
<code>\tl_put_right:co</code>

`\tl_put_right:Nn $\langle tl\ var.\rangle$ {\mathit{token list}}`

These functions append $\langle token\ list\rangle$ to the right of $\langle tl\ var.\rangle$.

<code>\tl_gput_left:Nn</code>
<code>\tl_gput_left:No</code>
<code>\tl_gput_left:NV</code>
<code>\tl_gput_left:Nx</code>
<code>\tl_gput_left:cn</code>
<code>\tl_gput_left:co</code>
<code>\tl_gput_left:cV</code>

`\tl_gput_left:Nn $\langle tl\ var.\rangle$ {\mathit{token list}}`

These functions will append $\langle token\ list\rangle$ globally to the left of $\langle tl\ var.\rangle$.

<code>\tl_gput_right:Nn</code>
<code>\tl_gput_right:No</code>
<code>\tl_gput_right:NV</code>
<code>\tl_gput_right:Nx</code>
<code>\tl_gput_right:cn</code>
<code>\tl_gput_right:co</code>
<code>\tl_gput_right:cV</code>

`\tl_gput_right:Nn $\langle tl\ var.\rangle$ {\mathit{token list}}`

These functions will globally append $\langle token\ list\rangle$ to the right of $\langle tl\ var.\rangle$.

A word of warning is appropriate here: Token list variables are implemented as macros and as such currently inherit some of the peculiarities of how T_EX handles #s in the argument of macros. In particular, the following actions are legal

```

\tl_set:Nn \l_tmpa_tl{##1}
\tl_put_right:Nn \l_tmpa_tl{##2}
\tl_set:No \l_tmpb_tl{\l_tmpa_tl ##3}

```

x type expansions where macros being expanded contain #s do not work and will not work until there is an `\expanded` primitive in the engine. If you want them to work you must double #s another level.

<code>\tl_set_eq:NN</code>
<code>\tl_set_eq:Nc</code>
<code>\tl_set_eq:cN</code>
<code>\tl_set_eq:cc</code>
<code>\tl_gset_eq:NN</code>
<code>\tl_gset_eq:Nc</code>
<code>\tl_gset_eq:cN</code>
<code>\tl_gset_eq:cc</code>

```

\tl_set_eq:NN <tl var. 1> <tl var. 2>

```

Fast form for `\tl_set:No <tl var. 1> {\<tl var. 2>}`
when `<tl var. 2>` is known to be a variable of type `tl`.

<code>\tl_to_str:N</code>
<code>\tl_to_str:c</code>

```

\tl_to_str:N <tl var.>

```

This function returns the token list kept in `<tl var.>` as a string list with all characters catcoded to ‘other’.

<code>\tl_to_str:n</code>

```

\tl_to_str:n {\<token list>}

```

This function turns its argument into a string where all characters have catcode ‘other’.

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

<code>\tl_rescan:nn</code>

```

\tl_rescan:nn {\<catcode setup>} {\<token list>}

```

Returns the result of re-tokenising `<token list>` with the catcode setup (and whatever other redefinitions) specified. This is useful because the catcodes of characters are ‘frozen’ when first tokenised; this allows their meaning to be changed even after they’ve been read as an argument. Also see `\tl_set_rescan:Nnn` below.

TeXhackers note: This is a wrapper around ε -TeX’s `\scantokens`.

<code>\tl_set_rescan:Nnn</code> <code>\tl_set_rescan:Nno</code> <code>\tl_set_rescan:Nnx</code> <code>\tl_gset_rescan:Nnn</code> <code>\tl_gset_rescan:Nno</code> <code>\tl_gset_rescan:Nnx</code>	<code>\tl_set_rescan:Nnn <tl var.> {<catcode setup>} {<token list>}</code>
---	--

Sets $\langle tl\ var.\rangle$ to the result of re-tokenising $\langle token\ list\rangle$ with the catcode setup (and whatever other redefinitions) specified.

TeXhackers note: This is a wrapper around ε -TeX's `\scantokens`.

43 Predicates and conditionals

<code>\tl_if_empty_p:N *</code> <code>\tl_if_empty_p:c *</code>	<code>\tl_if_empty_p:N <tl var.></code>
--	---

This predicate returns ‘true’ if $\langle tl\ var.\rangle$ is ‘empty’ i.e., doesn’t contain any tokens.

<code>\tl_if_empty:NTF *</code> <code>\tl_if_empty:cTF *</code>	<code>\tl_if_empty:NTF <tl var.> {<true code>} {<false code>}</code>
--	--

Execute $\langle true\ code\rangle$ if $\langle tl\ var.\rangle$ is empty and $\langle false\ code\rangle$ if it contains any tokens.

<code>\tl_if_eq_p:NN *</code> <code>\tl_if_eq_p:cN *</code> <code>\tl_if_eq_p:Nc *</code> <code>\tl_if_eq_p:cc *</code>	<code>\tl_if_eq_p:NN <tl var.₁> <tl var.₂></code>
--	---

Predicate function which returns ‘true’ if the two token list variables are identical and ‘false’ otherwise.

<code>\tl_if_eq:NNTF *</code> <code>\tl_if_eq:cNTF *</code> <code>\tl_if_eq:NcTF *</code> <code>\tl_if_eq:ccTF *</code>	<code>\tl_if_eq:NNTF <tl var.₁> <tl var.₂> {<true code>} {<false code>}</code>
--	--

Execute $\langle true\ code\rangle$ if $\langle tl\ var.₁\rangle$ holds the same token list as $\langle tl\ var.₂\rangle$ and $\langle false\ code\rangle$ otherwise.

<code>\tl_if_empty_p:n *</code> <code>\tl_if_empty_p:V *</code> <code>\tl_if_empty_p:o *</code> <code>\tl_if_empty:nTF *</code> <code>\tl_if_empty:VTF *</code> <code>\tl_if_empty:oTF *</code>	<code>\tl_if_empty:nTF {<token list>} {<true code>} {<false code>}</code>
--	---

Execute $\langle true\ code\rangle$ if $\langle token\ list\rangle$ doesn't contain any tokens and $\langle false\ code\rangle$ otherwise.

$\backslash\mathrm{tl_if_eq:nnTF}$	$\backslash\mathrm{tl_if_eq:nnTF}\ \langle token\ list1\rangle\ \{\langle token\ list2\rangle\}\ \{\langle true\ code\rangle\}\ \{\langle false\ code\rangle\}$
--------------------------------------	---

Tests if $\langle token\ list1\rangle$ and $\langle token\ list2\rangle$ both in respect of character codes and category codes. Either the $\langle true\ code\rangle$ or $\langle false\ code\rangle$ in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

$\backslash\mathrm{tl_if_blank_p:n}$	*	
$\backslash\mathrm{tl_if_blank:nTF}$	*	
$\backslash\mathrm{tl_if_blank_p:VTF}$	*	
$\backslash\mathrm{tl_if_blank_p:oTF}$	*	
$\backslash\mathrm{tl_if_blank:VTF}$	*	
$\backslash\mathrm{tl_if_blank:oTF}$	*	

$\backslash\mathrm{tl_if_blank:nTF}\ \{\langle token\ list\rangle\}\ \{\langle true\ code\rangle\}\ \{\langle false\ code\rangle\}$

Execute $\langle true\ code\rangle$ if $\langle token\ list\rangle$ is blank meaning that it is either empty or contains only blank spaces.

$\backslash\mathrm{tl_if_single_p:n}$	*	
$\backslash\mathrm{tl_if_single:nTF}$	*	
$\backslash\mathrm{tl_if_single_p:NTF}$	*	
$\backslash\mathrm{tl_if_single:NTF}$	*	

$\backslash\mathrm{tl_if_single:NTF}\ \{\langle tl\ var.\rangle\}\ \{\langle true\ code\rangle\}\ \{\langle false\ code\rangle\}$
 $\backslash\mathrm{tl_if_single:nTF}\ \{\langle token\ list\rangle\}\ \{\langle true\ code\rangle\}\ \{\langle false\ code\rangle\}$

Conditional returning true if the token list or the contents of the `tl var.` consists of a single token only.

Note that an input of ‘space’⁶ returns $\langle true\rangle$ from this function.

$\backslash\mathrm{tl_to_lowercase:n}$	
$\backslash\mathrm{tl_to_uppercase:n}$	

$\backslash\mathrm{tl_to_lowercase:n}\ \{\langle token\ list\rangle\}$

$\backslash\mathrm{tl_to_lowercase:n}$ converts all tokens in $\langle token\ list\rangle$ to their lower case representation. Similar for $\backslash\mathrm{tl_to_uppercase:n}$.

T_EXhackers note: These are the T_EX primitives `\lowercase` and `\uppercase` renamed.

44 Working with the contents of token lists

$\backslash\mathrm{tl_map_function:nN}$	*	
$\backslash\mathrm{tl_map_function:NN}$	*	
$\backslash\mathrm{tl_map_function:cN}$	*	

$\backslash\mathrm{tl_map_function:nN}\ \{\langle token\ list\rangle\}\ \langle function\rangle$
 $\backslash\mathrm{tl_map_function:NN}\ \langle tl\ var.\rangle\ \langle function\rangle$

⁶But remember any number of consecutive spaces are read as a single space by T_EX.

Runs through all elements in a $\langle token list \rangle$ from left to right and places $\langle function \rangle$ in front of each element. As this function will also pick up elements in brace groups, the element is returned with braces and hence $\langle function \rangle$ should be a function with a `:n` suffix even though it may very well only deal with a single token.

This function uses a purely expandable loop function and will stay so as long as $\langle function \rangle$ is expandable too.

<code>\tl_map_inline:nn</code>	
<code>\tl_map_inline:Nn</code>	<code>\tl_map_inline:nn {\langle token list \rangle} {\langle inline function \rangle}</code>
<code>\tl_map_inline:cn</code>	<code>\tl_map_inline:Nn \langle tl var. \rangle {\langle inline function \rangle}</code>

Allows a syntax like `\tl_map_inline:nn {\langle token list \rangle} {\token_to_str:N ##1}`. This renders it non-expandable though. Remember to double the `#`s for each level.

<code>\tl_map_variable:nNn</code>	
<code>\tl_map_variable:NNn</code>	<code>\tl_map_variable:nNn {\langle token list \rangle} \langle temp \rangle {\langle action \rangle}</code>
<code>\tl_map_variable:cNn</code>	<code>\tl_map_variable:NNn \langle tl var. \rangle \langle temp \rangle {\langle action \rangle}</code>

Assigns $\langle temp \rangle$ to each element on $\langle token list \rangle$ and executes $\langle action \rangle$. As there is an assignment in this process it is not expandable.

TeXhackers note: This is the L^AT_EX2 function `\tfor` but with a more sane syntax. Also it works by tail recursion and so is faster as lists grow longer.

<code>\tl_map_break:</code>	<code>\tl_map_break:</code>
-----------------------------	-----------------------------

For breaking out of a loop. Must not be nested inside a primitive `\if` structure.

<code>\tl_reverse:n</code>	
<code>\tl_reverse:V</code>	
<code>\tl_reverse:o</code>	<code>\tl_reverse:n {\langle token_1 \rangle \langle token_2 \rangle \dots \langle token_n \rangle}</code>
<code>\tl_reverse:N</code>	<code>\tl_reverse:N \langle tl var. \rangle</code>

Reverse the token list (or the token list in the $\langle tl var. \rangle$) to result in $\langle token_n \rangle \dots \langle token_2 \rangle \langle token_1 \rangle$. Note that spaces in this token list are gobbled in the process.

Note also that braces are lost in the process of reversing a $\langle tl var. \rangle$. That is, `\tl_set:Nn \l_tmpa_tl {a{bcd}e} \tl_reverse:N \l_tmpa_tl` will result in `ebcda`. This behaviour is probably more of a bug than a feature.

<code>\tl_elt_count:n *</code>	
<code>\tl_elt_count:V *</code>	
<code>\tl_elt_count:o *</code>	<code>\tl_elt_count:n {\langle token list \rangle}</code>
<code>\tl_elt_count:N *</code>	<code>\tl_elt_count:N \langle tl var. \rangle</code>

Returns the number of elements in the token list. Brace groups encountered count as one element. Note that spaces in this token list are gobbled in the process.

45 Variables and constants

`\c_job_name_tl` Constant that gets the ‘job name’ assigned when T_EX starts.

T_EXhackers note: This is the new name for the primitive `\jobname`. It is a constant that is set by T_EX and should not be overwritten by the package.

`\c_empty_tl` Constant that is always empty.

T_EXhackers note: This was named `\@empty` in L^AT_EX2 and `\empty` in plain T_EX.

`\c_space_tl` A space token contained in a token list (compare this with `\char_space_token`). For use where an explicit space is required.

`\l_tmpa_tl`
`\l_tmpb_tl`
`\g_tmpa_tl`
`\g_tmpb_tl` Scratch register for immediate use. They are not used by conditionals or predicate functions. However, it is important to note that you should never rely on such scratch variables unless you fully control the code used between setting them and retrieving their value. Calling code from other modules, or worse allowing arbitrary user input to interfere might result in them not containing what you expect. In that is the case you better define your own scratch variables that are tight to your code by giving them suitable names.

`\l_tl_replace_tl` Internal register used in the replace functions.

`\l_kernel_testa_tl`
`\l_kernel_testb_tl` Registers used for conditional processing if the engine doesn’t support arbitrary string comparison. Not for use outside the kernel code!

`\l_kernel_tmpa_tl`
`\l_kernel_tmpb_tl` Scratch registers reserved for other places in kernel code. Not for use outside the kernel code!

`\g_tl_inline_level_int` Internal register used in the inline map functions.

46 Searching for and replacing tokens

<code>\tl_if_in:NnTF</code>	\tl_if_in:NnTF <tl var.> {<item>} {<true code>} {<false code>}
<code>\tl_if_in:cnTF</code>	
<code>\tl_if_in:nnTF</code>	
<code>\tl_if_in:VnTF</code>	
<code>\tl_if_in:onTF</code>	

Function that tests if $\langle item \rangle$ is in $\langle tl\ var. \rangle$. Depending on the result either $\langle true\ code \rangle$ or $\langle false\ code \rangle$ is executed. Note that $\langle item \rangle$ cannot contain brace groups nor $\#_6$ tokens.

<code>\tl_replace_in:Nnn</code>	\tl_replace_in:Nnn <tl var.> {<item ₁ >} {<item ₂ >}
<code>\tl_replace_in:cnn</code>	
<code>\tl_greplace_in:Nnn</code>	
<code>\tl_greplace_in:cnn</code>	

Replaces the leftmost occurrence of $\langle item_1 \rangle$ in $\langle tl\ var. \rangle$ with $\langle item_2 \rangle$ if present, otherwise the $\langle tl\ var. \rangle$ is left untouched. Note that $\langle item_1 \rangle$ cannot contain brace groups nor $\#_6$ tokens, and $\langle item_2 \rangle$ cannot contain $\#_6$ tokens.

<code>\tl_replace_all_in:Nnn</code>	\tl_replace_all_in:Nnn <tl var.> {<item ₁ >} {<item ₂ >}
<code>\tl_replace_all_in:cnn</code>	
<code>\tl_greplace_all_in:Nnn</code>	
<code>\tl_greplace_all_in:cnn</code>	

Replaces *all* occurrences of $\langle item_1 \rangle$ in $\langle tl\ var. \rangle$ with $\langle item_2 \rangle$. Note that $\langle item_1 \rangle$ cannot contain brace groups nor $\#_6$ tokens, and $\langle item_2 \rangle$ cannot contain $\#_6$ tokens.

<code>\tl_remove_in:Nn</code>	\tl_remove_in:Nn <tl var.> {<item>}
<code>\tl_remove_in:cn</code>	
<code>\tl_gremove_in:Nn</code>	
<code>\tl_gremove_in:cn</code>	

Removes the leftmost occurrence of $\langle item \rangle$ from $\langle tl\ var. \rangle$ if present. Note that $\langle item \rangle$ cannot contain brace groups nor $\#_6$ tokens.

<code>\tl_remove_all_in:Nn</code>	\tl_remove_all_in:Nn <tl var.> {<item>}
<code>\tl_remove_all_in:cn</code>	
<code>\tl_gremove_all_in:Nn</code>	
<code>\tl_gremove_all_in:cn</code>	

Removes *all* occurrences of $\langle item \rangle$ from $\langle tl\ var. \rangle$. Note that $\langle item \rangle$ cannot contain brace groups nor $\#_6$ tokens.

47 Heads or tails?

Here are some functions for grabbing either the head or tail of a list and perform some tests on it.

<code>\tl_head:n</code>	<code>*</code>	
<code>\tl_head:V</code>	<code>*</code>	
<code>\tl_head:v</code>	<code>*</code>	
<code>\tl_tail:n</code>	<code>*</code>	
<code>\tl_tail:V</code>	<code>*</code>	
<code>\tl_tail:v</code>	<code>*</code>	
<code>\tl_tail:f</code>	<code>*</code>	
<code>\tl_head_i:n</code>	<code>*</code>	
<code>\tl_head_iii:n</code>	<code>*</code>	
<code>\tl_head_iii:f</code>	<code>*</code>	
<code>\tl_head:w</code>	<code>*</code>	
<code>\tl_tail:w</code>	<code>*</code>	
<code>\tl_head_i:w</code>	<code>*</code>	<code>\tl_head:n { <token₁><token₂>...<token_k> }</code>
<code>\tl_head_iii:w</code>	<code>*</code>	<code>\tl_tail:n { <token₁><token₂>...<token_k> }</code>
		<code>\tl_head:w <token₁><token₂>...<token_k> \q_stop</code>

These functions return either the head or the tail from a list of tokens, thus in the above example `\tl_head:n` would return `<token1>` and `\tl_tail:n` would return `<token2>...<tokenk>`. `\tl_head_iii:n` returns the first three tokens. The `:w` versions require some care as they expect the token list to be delimited by `\q_stop`.

T_EXhackers note: These are the Lisp functions `car` and `cdr` but with L^AT_EX3 names.

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

Returns `<true>` if the first token in `<token list>` is equal to `<token>` and `<false>` otherwise. The `meaning` version compares the two tokens with `\if_meaning:w`.

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

Returns `<true>` if the first token in `<token list>` is equal to `<token>` and `<false>` otherwise. The `meaning` version compares the two tokens with `\if_charcode:w` but it prevents expansion of them. If you want them to expand, you can use an `f` type expansion first

(define `\tl_if_head_eq_charcode:fNTF` or similar).

<code>\tl_if_head_eq_catcode:p:nN</code> \star <code>\tl_if_head_eq_catcode:nNTF</code> \star	<code>\tl_if_head_eq_catcode:nNTF</code> $\{\langle token\ list\rangle\}$ $\langle token\rangle$ $\{\langle true\rangle\}$ $\{\langle false\rangle\}$
--	--

Returns $\langle true\rangle$ if the first token in $\langle token\ list\rangle$ is equal to $\langle token\rangle$ and $\langle false\rangle$ otherwise. This version uses `\if_catcode:w` for the test but is otherwise identical to the `charcode` version.

Part XI

The l3toks package

Token Registers

There is a second form beside token list variables in which L^AT_EX3 stores token lists, namely the internal T_EX token registers. Functions dealing with these registers got the prefix `\toks_`. Unlike token list variables we have an accessing function as one can see below.

The main difference between $\langle toks\rangle$ (token registers) and $\langle tl\ var.\rangle$ (token list variable) is their behavior regarding expansion. While $\langle tl\ vars\rangle$ expand fully (i.e., until only unexpandable tokens are left) inside an argument that is subject to expansion (i.e., denoted by \mathbf{x}) $\langle toks\rangle$'s expand always only up to one level, i.e., passing their contents without further expansion.

There are fewer restrictions on the contents of a token register over a token list variable. So while $\langle token\ list\rangle$ is used to describe the contents of both of these, bear in mind that slightly different lists of tokens are allowed in each case. The best (only?) example is that a $\langle toks\rangle$ can contain the `#` character (i.e., characters of catcode 6), whereas a $\langle tl\ var.\rangle$ will require its input to be sanitised before that is possible.

If you're not sure which to use between a $\langle tl\ var.\rangle$ or a $\langle toks\rangle$, consider what data you're trying to hold. If you're dealing with function parameters involving `#`, or building some sort of data structure then you probably want a $\langle toks\rangle$ (e.g., `l3prop` uses $\langle toks\rangle$ to store its property lists).

If you're storing ad-hoc data for later use (possibly from direct user input) then usually a $\langle tl\ var.\rangle$ will be what you want.

48 Allocation and use

<code>\toks_new:N</code> <code>\toks_new:c</code>	<code>\toks_new:N</code> $\langle toks\rangle$
--	--

Defines $\langle toks \rangle$ to be a new token list register.

T_EXhackers note: This is the L^AT_EX3 allocation for what was called `\newtoks` in plain T_EX.

<code>\toks_use:N</code> <code>\toks_use:c</code>	<code>\toks_use:N $\langle toks \rangle$</code>
--	--

Accesses the contents of $\langle toks \rangle$. Contrary to token list variables $\langle toks \rangle$ can't be access simply by calling them directly.

T_EXhackers note: Something like `\the $\langle toks \rangle$` .

<code>\toks_set:Nn</code> <code>\toks_set:NV</code> <code>\toks_set:Nv</code> <code>\toks_set:No</code> <code>\toks_set:Nx</code> <code>\toks_set:Nf</code> <code>\toks_set:cn</code> <code>\toks_set:co</code> <code>\toks_set:cV</code> <code>\toks_set:cv</code> <code>\toks_set:cx</code> <code>\toks_set:cf</code>	<code>\toks_set:Nn $\langle toks \rangle$ {$\langle token list \rangle$}</code>
--	---

Defines $\langle toks \rangle$ to hold the token list $\langle token list \rangle$.

T_EXhackers note: `\toks_set:Nn` could have been specified in plain T_EX by $\langle toks \rangle = \{ \langle token list \rangle \}$ but all other functions have no counterpart in plain T_EX.

<code>\toks_gset:Nn</code> <code>\toks_gset:NV</code> <code>\toks_gset:No</code> <code>\toks_gset:Nx</code> <code>\toks_gset:cn</code> <code>\toks_gset:cV</code> <code>\toks_gset:co</code> <code>\toks_gset:cx</code>	<code>\toks_gset:Nn $\langle toks \rangle$ {$\langle token list \rangle$}</code>
--	--

Defines $\langle toks \rangle$ to globally hold the token list $\langle token list \rangle$.

<code>\toks_set_eq:NN</code> <code>\toks_set_eq:cN</code> <code>\toks_set_eq:Nc</code> <code>\toks_set_eq:cc</code>	<code>\toks_set_eq:NN $\langle toks_1 \rangle$ $\langle toks_2 \rangle$</code>
--	--

Set $\langle toks_1 \rangle$ to the value of $\langle toks_2 \rangle$. Don't try to use `\toks_set:Nn` for this purpose if the second argument is also a token register.

<code>\toks_gset_eq:NN</code>
<code>\toks_gset_eq:cN</code>
<code>\toks_gset_eq:Nc</code>
<code>\toks_gset_eq:cc</code>

`\toks_gset_eq:NN $\langle toks_1 \rangle$ $\langle toks_2 \rangle$`

The $\langle toks_1 \rangle$ globally set to the value of $\langle toks_2 \rangle$. Don't try to use `\toks_gset:Nn` for this purpose if the second argument is also a token register.

<code>\toks_clear:N</code>
<code>\toks_clear:c</code>
<code>\toks_gclear:N</code>
<code>\toks_gclear:c</code>

`\toks_clear:N $\langle toks \rangle$`

The $\langle toks \rangle$ is locally or globally cleared.

<code>\toks_use_clear:N</code>
<code>\toks_use_clear:c</code>
<code>\toks_use_gclear:N</code>
<code>\toks_use_gclear:c</code>

`\toks_use_clear:N $\langle toks \rangle$`

Accesses the contents of $\langle toks \rangle$ and clears (locally or globally) it afterwards. Actually the clearing operation is done in a way that does not prohibit the access of the following tokens in the input stream with functions stored in the token register. In other words this function is not exactly the same as calling `\toks_use:N $\langle toks \rangle$ \toks_clear:N $\langle toks \rangle$` in sequence.

<code>\toks_show:N</code>
<code>\toks_show:c</code>

`\toks_show:N $\langle toks \rangle$`

Displays the contents of $\langle toks \rangle$ in the terminal output and log file. # signs in the $\langle toks \rangle$ will be shown doubled.

T_EXhackers note: Something like `\showthe $\langle toks \rangle$` .

49 Adding to the contents of token registers

<code>\toks_put_left:Nn</code>
<code>\toks_put_left:NV</code>
<code>\toks_put_left:No</code>
<code>\toks_put_left:Nx</code>
<code>\toks_put_left:cn</code>
<code>\toks_put_left:cV</code>
<code>\toks_put_left:co</code>

`\toks_put_left:Nn <toks> {<token list>}`

These functions will append *<token list>* to the left of *<toks>*. Assignment is done locally. If possible append to the right since this operation is faster.

<code>\toks_gput_left:Nn</code>
<code>\toks_gput_left:NV</code>
<code>\toks_gput_left:No</code>
<code>\toks_gput_left:Nx</code>
<code>\toks_gput_left:cn</code>
<code>\toks_gput_left:cV</code>
<code>\toks_gput_left:co</code>

`\toks_gput_left:Nn <toks> {<token list>}`

These functions will append *<token list>* to the left of *<toks>*. Assignment is done globally. If possible append to the right since this operation is faster.

<code>\toks_put_right:Nn</code>
<code>\toks_put_right:NV</code>
<code>\toks_put_right:No</code>
<code>\toks_put_right:Nx</code>
<code>\toks_put_right:cV</code>
<code>\toks_put_right:cn</code>
<code>\toks_put_right:co</code>

`\toks_put_right:Nn <toks> {<token list>}`

These functions will append *<token list>* to the right of *<toks>*. Assignment is done locally.

<code>\toks_put_right:Nf</code>

`\toks_put_right:Nf <toks> {<token list>}`

Variant of the above. `:Nf` is used by `template.dtx` and will perhaps be moved to that package.

<code>\toks_gput_right:Nn</code>
<code>\toks_gput_right:NV</code>
<code>\toks_gput_right:No</code>
<code>\toks_gput_right:Nx</code>
<code>\toks_gput_right:cn</code>
<code>\toks_gput_right:cV</code>
<code>\toks_gput_right:co</code>

`\toks_gput_right:Nn <toks> {<token list>}`

These functions will append $\langle token\ list \rangle$ to the right of $\langle toks \rangle$. Assignment is done globally.

50 Predicates and conditionals

<code>\toks_if_empty_p:N</code>	<code>*</code>
<code>\toks_if_empty:N\underline{TF}</code>	<code>*</code>
<code>\toks_if_empty_p:c\underline{TF}</code>	<code>*</code>
<code>\toks_if_empty:c\underline{TF}</code>	<code>*</code>

$$\backslash\toks_if_empty:N\mathit{TF}\ \langle toks \rangle\ \{\langle true\ code \rangle\}\ \{\langle false\ code \rangle\}$$

Expandable test for whether $\langle toks \rangle$ is empty.

<code>\toks_if_eq:N\underline{NTF}</code>	<code>*</code>
<code>\toks_if_eq:Nc\underline{TF}</code>	<code>*</code>
<code>\toks_if_eq:cN\underline{TF}</code>	<code>*</code>
<code>\toks_if_eq:cc\underline{TF}</code>	<code>*</code>
<code>\toks_if_eq_p:N\underline{NTF}</code>	<code>*</code>
<code>\toks_if_eq_p:cN\underline{TF}</code>	<code>*</code>
<code>\toks_if_eq_p:Nc\underline{TF}</code>	<code>*</code>
<code>\toks_if_eq_p:cc\underline{TF}</code>	<code>*</code>

$$\backslash\toks_if_eq:N\mathit{NTF}\ \langle toks_1 \rangle\ \langle toks_2 \rangle\ \{\langle true\ code \rangle\}\ \{\langle false\ code \rangle\}$$

Expandably tests if $\langle toks_1 \rangle$ and $\langle toks_2 \rangle$ are equal.

51 Variable and constants

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

<code>\l_tmpa_toks</code>	Scratch register for immediate use. They are not used by conditionals or predicate functions.
<code>\l_tmpb_toks</code>	
<code>\l_tmpc_toks</code>	
<code>\g_tmpa_toks</code>	
<code>\g_tmpb_toks</code>	
<code>\g_tmpc_toks</code>	

<code>\l_tl_replace_toks</code>	A placeholder for contents of functions replacing contents of strings.
---------------------------------	--

Part XII

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.

52 Creating and initialising sequences

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

`\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.

<code>\seq_clear:N</code>
<code>\seq_clear:c</code>

`\seq_clear:N <sequence>`

Clears all items from the *⟨sequence⟩* within the scope of the current T_EX group.

<code>\seq_gclear:N</code>
<code>\seq_gclear:c</code>

`\seq_gclear:N <sequence>`

Clears all entries from the *⟨sequence⟩* globally.

<code>\seq_clear_new:N</code>
<code>\seq_clear_new:c</code>

`\seq_clear_new:N <sequence>`

If the *⟨sequence⟩* already exists, clears it within the scope of the current T_EX group. If the *⟨sequence⟩* is not defined, it will be created (using `\seq_new:N`). Thus the sequence is guaranteed to be available and clear within the current T_EX group. The *⟨sequence⟩* will exist globally, but the content outside of the current T_EX group is not specified.

<code>\seq_gclear_new:N</code>
<code>\seq_gclear_new:c</code>

`\seq_gclear_new:N <sequence>`

If the *⟨sequence⟩* already exists, clears it globally. If the *⟨sequence⟩* is not defined, it will be created (using `\seq_new:N`). Thus the sequence is guaranteed to be available and globally clear.

<code>\seq_set_eq:NN</code>
<code>\seq_set_eq:cN</code>
<code>\seq_set_eq:Nc</code>
<code>\seq_set_eq:cc</code>

`\seq_set_eq:NN <sequence1> <sequence2>`

Sets the content of $\langle sequence1 \rangle$ equal to that of $\langle sequence2 \rangle$. This assignment is restricted to the current \TeX group level.

<code>\seq_gset_eq:NN</code>
<code>\seq_gset_eq:cN</code>
<code>\seq_gset_eq:Nc</code>
<code>\seq_gset_eq:cc</code>

`\seq_gset_eq:NN $\langle sequence1 \rangle$ $\langle sequence2 \rangle$`

Sets the content of $\langle sequence1 \rangle$ equal to that of $\langle sequence2 \rangle$. This assignment is global and so is not limited by the current \TeX group level.

<code>\seq_concat:NNN</code>
<code>\seq_concat:ccc</code>

`\seq_concat:NNN $\langle sequence1 \rangle$ $\langle sequence2 \rangle$ $\langle sequence3 \rangle$`

Concatenates the content of $\langle sequence2 \rangle$ and $\langle sequence3 \rangle$ together and saves the result in $\langle sequence1 \rangle$. The items in $\langle sequence2 \rangle$ will be placed at the left side of the new sequence. This operation is local to the current \TeX group and will remove any existing content in $\langle sequence1 \rangle$.

<code>\seq_gconcat:NNN</code>
<code>\seq_gconcat:ccc</code>

`\seq_gconcat:NNN $\langle sequence1 \rangle$ $\langle sequence2 \rangle$ $\langle sequence3 \rangle$`

Concatenates the content of $\langle sequence2 \rangle$ and $\langle sequence3 \rangle$ together and saves the result in $\langle sequence1 \rangle$. The items in $\langle sequence2 \rangle$ will be placed at the left side of the new sequence. This operation is global and will remove any existing content in $\langle sequence1 \rangle$.

53 Appending data to sequences

<code>\seq_put_left:Nn</code>
<code>\seq_put_left:NV</code>
<code>\seq_put_left:Nv</code>
<code>\seq_put_left:No</code>
<code>\seq_put_left:Nx</code>
<code>\seq_put_left:cn</code>
<code>\seq_put_left:cV</code>
<code>\seq_put_left:cv</code>
<code>\seq_put_left:co</code>
<code>\seq_put_left:cx</code>

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

Appends the $\langle item \rangle$ to the left of the $\langle sequence \rangle$. The assignment is restricted to the

current \TeX group.

<code>\seq_gput_left:Nn</code>
<code>\seq_gput_left:NV</code>
<code>\seq_gput_left:Nv</code>
<code>\seq_gput_left:No</code>
<code>\seq_gput_left:Nx</code>
<code>\seq_gput_left:cn</code>
<code>\seq_gput_left:cV</code>
<code>\seq_gput_left:cv</code>
<code>\seq_gput_left:co</code>
<code>\seq_gput_left:cx</code>

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

Appends the $\langle item \rangle$ to the left of the $\langle sequence \rangle$. The assignment is global.

<code>\seq_put_right:Nn</code>
<code>\seq_put_right:NV</code>
<code>\seq_put_right:Nv</code>
<code>\seq_put_right:No</code>
<code>\seq_put_right:Nx</code>
<code>\seq_put_right:cn</code>
<code>\seq_put_right:cV</code>
<code>\seq_put_right:cv</code>
<code>\seq_put_right:co</code>
<code>\seq_put_right:cx</code>

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

Appends the $\langle item \rangle$ to the right of the $\langle sequence \rangle$. The assignment is restricted to the current \TeX group.

<code>\seq_gput_right:Nn</code>
<code>\seq_gput_right:NV</code>
<code>\seq_gput_right:Nv</code>
<code>\seq_gput_right:No</code>
<code>\seq_gput_right:Nx</code>
<code>\seq_gput_right:cn</code>
<code>\seq_gput_right:cV</code>
<code>\seq_gput_right:cv</code>
<code>\seq_gput_right:co</code>
<code>\seq_gput_right:cx</code>

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

Appends the $\langle item \rangle$ to the right of the $\langle sequence \rangle$. The assignment is global.

54 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`.

<code>\seq_get_left:NN</code> <code>\seq_get_left:cN</code>	<code>\seq_get_left:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$
--	--

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 an error will be raised.

<code>\seq_get_right:NN</code> <code>\seq_get_right:cN</code>	<code>\seq_get_right:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$
--	---

Stores the right-most item from a $\langle sequence \rangle$ in the $\langle token\ list\ variable \rangle$ without removing it from the $\langle sequence \rangle$. The $\langle token\ list\ variable \rangle$ is assigned locally. If $\langle sequence \rangle$ is empty an error will be raised.

<code>\seq_pop_left:NN</code> <code>\seq_pop_left:cN</code>	<code>\seq_pop_left:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$
--	--

Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$, *i.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 an error will be raised.

<code>\seq_gpop_left:NN</code> <code>\seq_gpop_left:cN</code>	<code>\seq_gpop_left:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$
--	---

Pops the left-most item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$, *i.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 an error will be raised.

<code>\seq_pop_right:NN</code> <code>\seq_pop_right:cN</code>	<code>\seq_pop_right:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$
--	---

Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$, *i.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 an error will be raised.

<code>\seq_gpop_right:NN</code> <code>\seq_gpop_right:cN</code>	<code>\seq_gpop_right:NN</code> $\langle sequence \rangle$ $\langle token\ list\ variable \rangle$
--	--

Pops the right-most item from a $\langle sequence \rangle$ into the $\langle token\ list\ variable \rangle$, *i.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 an error will be raised.

55 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_remove_duplicates:N <sequence></code>
--	--

Removes duplicate items from the $\langle sequence \rangle$, leaving left most copy of each item in the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`. The removal is local to the current \TeX group.

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

<code>\seq_gremove_duplicates:N</code> <code>\seq_gremove_duplicates:c</code>	<code>\seq_gremove_duplicates:N <sequence></code>
--	---

Removes duplicate items from the $\langle sequence \rangle$, leaving left most copy of each item in the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`. The removal is applied globally.

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

<code>\seq_remove_all:Nn</code> <code>\seq_remove_all:cn</code>	<code>\seq_remove_all:Nn <sequence> {\item}</code>
--	--

Removes every occurrence of $\langle item \rangle$ from the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`. The removal is local to the current \TeX group.

<code>\seq_gremove_all:Nn</code> <code>\seq_gremove_all:cn</code>	<code>\seq_gremove_all:Nn <sequence> {\item}</code>
--	---

Removes each occurrence of $\langle item \rangle$ from the $\langle sequence \rangle$. The $\langle item \rangle$ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`. The removal is applied globally.

55.1 Sequence conditionals

<code>\seq_if_empty_p:N</code>	★	
<code>\seq_if_empty:N</code>	<u><i>TF</i></u>	★
<code>\seq_if_empty_p:c</code>	★	
<code>\seq_if_empty:c</code>	<u><i>TF</i></u>	★
		<code>\seq_if_empty_p:N</code> $\langle sequence \rangle$
		<code>\seq_if_empty:N</code> $\langle sequence \rangle$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$

Tests if the $\langle sequence \rangle$ is empty (containing no items). The branching versions then leave either $\langle true code \rangle$ or $\langle false code \rangle$ in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

<code>\seq_if_in:Nn</code>	<u><i>TF</i></u>
<code>\seq_if_in:NV</code>	<u><i>TF</i></u>
<code>\seq_if_in:Nv</code>	<u><i>TF</i></u>
<code>\seq_if_in:No</code>	<u><i>TF</i></u>
<code>\seq_if_in:Nx</code>	<u><i>TF</i></u>
<code>\seq_if_in:cn</code>	<u><i>TF</i></u>
<code>\seq_if_in:cV</code>	<u><i>TF</i></u>
<code>\seq_if_in:cv</code>	<u><i>TF</i></u>
<code>\seq_if_in:co</code>	<u><i>TF</i></u>
<code>\seq_if_in:cx</code>	<u><i>TF</i></u>
<code>\seq_if_in:NnTF</code> $\langle sequence \rangle$ $\{\langle item \rangle\}$ $\{\langle true code \rangle\}$ $\{\langle false code \rangle\}$	

Tests if the $\langle item \rangle$ is present in the $\langle sequence \rangle$. Either the $\langle true code \rangle$ or $\langle false code \rangle$ is left in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

56 Mapping to sequences

<code>\seq_map_function:NN</code>	★	
<code>\seq_map_function:cN</code>	★	
		<code>\seq_map_function:NN</code> $\langle sequence \rangle$ $\langle function \rangle$

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

<code>\seq_map_inline:Nn</code>	
<code>\seq_map_inline:cn</code>	
<code>\seq_map_inline:Nn</code>	$\langle sequence \rangle \{ \langle inline function \rangle \}$

Applies $\langle inline function \rangle$ to every $\langle item \rangle$ stored within the $\langle sequence \rangle$. The $\langle inline$

function) 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>\seq_map_variable:Nn</code>	
<code>\seq_map_variable:Ncn</code>	
<code>\seq_map_variable:cNn</code>	<code>\seq_map_variable:Nn $\langle sequence \rangle$</code>
<code>\seq_map_variable:ccn</code>	<code>$\langle tl var. \rangle$ $\{ \langle function using tl var. \rangle \}$</code>

Stores each entry in the $\langle sequence \rangle$ in turn in the $\langle tl var. \rangle$ and applies the $\langle function using tl var. \rangle$. The $\langle function \rangle$ will usually consist of code making use of the $\langle tl var. \rangle$, but this is not enforced. One variable mapping can be nested inside another. The $\langle items \rangle$ are returned from left to right. One variable mapping may be nested inside another.

<code>\seq_map_break: *</code>	<code>\seq_map_break:</code>
--------------------------------	------------------------------

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 low level T_EX errors.

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

<code>\seq_map_break:n *</code>	<code>\seq_map_break:n $\{ \langle tokens \rangle \}$</code>
---------------------------------	---

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 low level T_EX errors.

T_EXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\seq_break_point:n` before the *tokens* are inserted into the input stream. This will depend on the design of the mapping function.

56.1 Sequences as stacks

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

<code>\seq_get:NN</code>	<code>\seq_get:NN</code> <i>sequence</i> <i>token list variable</i>
<code>\seq_get:cN</code>	

Reads the top item from a *sequence* into the *token list variable* without removing it from the *sequence*. The *token list variable* is assigned locally. If *sequence* is empty an error will be raised.

<code>\seq_pop:NN</code>	<code>\seq_pop:NN</code> <i>sequence</i> <i>token list variable</i>
<code>\seq_pop:cN</code>	

Pops the top item from a *sequence* into the *token list variable*. Both of the variables are assigned locally. If *sequence* is empty an error will be raised.

<code>\seq_gpop:NN</code>	<code>\seq_gpop:NN</code> <i>sequence</i> <i>token list variable</i>
<code>\seq_gpop:cN</code>	

Pops the top item from a *sequence* into the *token list variable*. The *sequence* is modified globally, while the *token list variable* is assigned locally. If *sequence* is empty

the marker an error will be raised.

<code>\seq_push:Nn</code>
<code>\seq_push:NV</code>
<code>\seq_push:Nv</code>
<code>\seq_push:No</code>
<code>\seq_push:Nx</code>
<code>\seq_push:cn</code>
<code>\seq_push:cV</code>
<code>\seq_push:cv</code>
<code>\seq_push:co</code>
<code>\seq_push:cx</code>

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

Adds the $\{ \langle item \rangle \}$ to the top of the $\langle sequence \rangle$. The assignment is restricted to the current \TeX group.

<code>\seq_gpush:Nn</code>
<code>\seq_gpush:NV</code>
<code>\seq_gpush:Nv</code>
<code>\seq_gpush:No</code>
<code>\seq_gpush:Nx</code>
<code>\seq_gpush:cn</code>
<code>\seq_gpush:cV</code>
<code>\seq_gpush:cv</code>
<code>\seq_gpush:co</code>
<code>\seq_gpush:cx</code>

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

Pushes the $\langle item \rangle$ onto the end of the top of the $\langle sequence \rangle$. The assignment is global.

57 Viewing sequences

<code>\seq_show:N</code>
<code>\seq_show:c</code>

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

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

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 sequence. This gives an ordered list which can then be utilised with the `\clist_map_function:NN` function. Comma lists cannot contain empty items, thus

```
\clist_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.

58 Functions for creating/initialising comma-lists

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

`\clist_new:N <comma list>`

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

<code>\clist_set_eq:NN</code>
<code>\clist_set_eq:cN</code>
<code>\clist_set_eq:Nc</code>
<code>\clist_set_eq:cc</code>

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

Sets the content of *<comma list1>* equal to that of *<comma list2>*. This assignment is restricted to the current \TeX group level.

<code>\clist_gset_eq:NN</code>
<code>\clist_gset_eq:cN</code>
<code>\clist_gset_eq:Nc</code>
<code>\clist_gset_eq:cc</code>

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

Sets the content of *<comma list1>* equal to that of *<comma list2>*. This assignment is

global and so is not limited by the current T_EX group level.

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

`\clist_clear:N <comma list>`
 Clears all entries from the *<comma list>* within the scope of the current T_EX group.

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

`\clist_gclear:N <comma list>`
 Clears all entries from the *<comma list>* globally.

<code>\clist_clear_new:N</code>
<code>\clist_clear_new:c</code>
<code>\clist_gclear_new:N</code>
<code>\clist_gclear_new:c</code>

`\clist_clear_new:N <comma-list>`
 These functions locally or globally clear *<comma-list>* if it exists or otherwise allocates it.

59 Putting data in

<code>\clist_put_left:Nn</code>
<code>\clist_put_left:NV</code>
<code>\clist_put_left:No</code>
<code>\clist_put_left:Nx</code>
<code>\clist_put_left:cn</code>
<code>\clist_put_left:cV</code>
<code>\clist_put_left:co</code>

`\clist_put_left:Nn <comma list> {<entry>}`
 Adds *<entry>* onto the left of the *<comma list>*. The assignment is restricted to the current

TeX group.

```
\clist_gput_left:Nn
\clist_gput_left:NV
\clist_gput_left:No
\clist_gput_left:Nx
\clist_gput_left:cn
\clist_gput_left:cV
\clist_gput_left:co
```

```
\clist_gput_left:Nn <comma list> {\entry}
```

Adds $\langle entry \rangle$ onto the left of the $\langle comma list \rangle$. The assignment is global.

```
\clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:cV
\clist_put_right:co
```

```
\clist_put_right:Nn <comma list> {\entry}
```

Adds $\langle entry \rangle$ onto the right of the $\langle comma list \rangle$. The assignment is restricted to the current TeX group.

```
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:co
```

```
\clist_gput_right:Nn <comma list> {\entry}
```

Adds $\langle entry \rangle$ onto the right of the $\langle comma list \rangle$. The assignment is global.

60 Getting data out

```
\clist_use:N ★
\clist_use:c ★
```

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

Recovers the content of a $\langle comma list \rangle$ and places it directly in the input stream. An error will be raised if the variable does not exist or if it is invalid. This function is intended mainly for use when a $\langle comma list \rangle$ is being saved to an auxiliary file.

```
\clist_show:N
\clist_show:c
```

```
\clist_show:N <clist>
```

Function that pauses the compilation and displays $\langle clist \rangle$ in the terminal output and in

the log file. (Usually used for diagnostic purposes.)

```
\clist_display:N
\clist_display:c \clist_display:N <comma list>
```

Displays the value of the *<comma list>* on the terminal.

```
\clist_get:NN
\clist_get:cN \clist_get:NN <comma-list> <tl var.>
```

Functions that locally assign the left-most item of *<comma-list>* to the token list variable *<tl var.>*. Item is not removed from *<comma-list>*! If you need a global return value you need to code something like this:

```
\clist_get:NN <comma-list> \l_tmpa_tl
\tl_gset_eq:NN <global tl var.> \l_tmpa_tl
```

But if this kind of construction is used often enough a separate function should be provided.

61 Mapping functions

We provide three types of mapping functions, each with their own strengths. The `\clist_map_function:NN` is expandable whereas `\clist_map_inline:Nn` type uses `##1` as a placeholder for the current item in *<clist>*. Finally we have the `\clist_map_variable:NNn` type which uses a user-defined variable as placeholder. Both the `_inline` and `_variable` versions are nestable.

```
\clist_map_function:NN *
\clist_map_function:Nc *
\clist_map_function:cN *
\clist_map_function:cc *
\clist_map_function:nN *
\clist_map_function:nc * \clist_map_function:NN <comma list> <function>
```

Applies *<function>* to every *<entry>* stored in the *<comma list>*. The *<function>* will receive one argument for each iteration. The *<entries>* in the *<comma list>* are supplied to the *<function>* reading from the left to the right. These function may be nested.

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

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

$\backslash\text{clist_map_variable:Nn}$ $\backslash\text{clist_map_variable:cNn}$ $\backslash\text{clist_map_variable:nNn}$	$\backslash\text{clist_map_variable:Nn} \langle comma\ list \rangle \langle temp\text{-}var \rangle \{ \langle action \rangle \}$
--	---

Assigns $\langle temp\text{-}var \rangle$ to each element in $\langle clist \rangle$ and then executes $\langle action \rangle$ which should contain $\langle temp\text{-}var \rangle$. As the operation performs an assignment, it is not expandable.

T_EXhackers note: These functions resemble the L^AT_EX 2_ε function $\backslash\text{@for}$ but does not borrow the somewhat strange syntax.

$\backslash\text{clist_map_break:} \star$	$\backslash\text{clist_map_break:}$
---	---------------------------------------

Used to terminate a $\backslash\text{clist_map_}\dots$ function before all entries in the $\langle comma\ list \rangle$ 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 $\backslash\text{clist_map_}\dots$ scenario will lead low level T_EX errors.

62 Predicates and conditionals

$\backslash\text{clist_if_empty_p:N} \star$ $\backslash\text{clist_if_empty_p:c} \star$ $\backslash\text{clist_if_empty:N}\underline{TF} \star$ $\backslash\text{clist_if_empty:c}\underline{TF} \star$	$\backslash\text{clist_if_empty_p:N} \langle clist \rangle$ $\backslash\text{clist_if_empty:N}\underline{TF} \langle clist \rangle \{ \langle true\ code \rangle \} \{ \langle false\ code \rangle \}$
--	--

Tests if the $\langle comma\ list \rangle$ is empty (containing no items). The branching versions then leave either $\langle true\ code \rangle$ or $\langle false\ code \rangle$ in the input stream, as appropriate to the truth

of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

<code>\clist_if_eq_p:NN</code>	<code>*</code>	
<code>\clist_if_eq_p:Nc</code>	<code>*</code>	
<code>\clist_if_eq_p:cN</code>	<code>*</code>	
<code>\clist_if_eq_p:cc</code>	<code>*</code>	
<code>\clist_if_eq:NNTF</code>	<code>*</code>	
<code>\clist_if_eq:NcTF</code>	<code>*</code>	
<code>\clist_if_eq:cNTF</code>	<code>*</code>	
<code>\clist_if_eq:ccTF</code>	<code>*</code>	
		<code>\clist_if_eq_p:NN {<clist₁>} {<clist₂>}</code>
		<code>\clist_if_eq:NNTF {<clist₁>} {<clist₂>} {<true code>}</code>
		<code>{<false code>}</code>

Compares the content of two *<comma lists>* and is logically **true** if the two contain the same list of entries in the same order. The branching versions then leave either *<true code>* or *<false code>* in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

<code>\clist_if_in:NnTF</code>		
<code>\clist_if_in:NVTF</code>		
<code>\clist_if_in:NoTF</code>		
<code>\clist_if_in:cnTF</code>		
<code>\clist_if_in:cVTF</code>		
<code>\clist_if_in:coTF</code>		
		<code>\clist_if_in:NnTF <clist> {<entry>} {<true code>}</code>
		<code>{<false code>}</code>

Tests if the *<entry>* is present in the *<comma list>* as a discrete entry. The *<entry>* cannot contain the tokens `{`, `}` or `#` (assuming the usual \TeX category codes apply). Either the *<true code>* or *<false code>* in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

63 Higher level functions

<code>\clist_concat:NNN</code>		
<code>\clist_concat:ccc</code>		
		<code>\clist_concat:NNN <clist1> <clist2> <clist3></code>

Concatenates the content of *<comma list2>* and *<comma list3>* together and saves the result in *<comma list1>*. *<comma list2>* will be placed at the left side of the new comma list. This operation is local to the current \TeX group and will remove any existing content in *<comma list1>*.

<code>\clist_gconcat:NNN</code>		
<code>\clist_gconcat:ccc</code>		
		<code>\clist_gconcat:NNN <clist1> <clist2> <clist3></code>

Concatenates the content of *<comma list2>* and *<comma list3>* together and saves the

result in $\langle comma list1 \rangle$. $\langle comma list2 \rangle$ will be placed at the left side of the new comma list. This operation is global and will remove any existing content in $\langle comma list1 \rangle$.

`\clist_remove_duplicates:N` `\clist_remove_duplicates:N` $\langle comma list \rangle$

Removes duplicate entries from the $\langle comma list \rangle$, leaving left most entry in the $\langle comma list \rangle$. The removal is local to the current \TeX group.

`\clist_gremove_duplicates:N` `\clist_gremove_duplicates:N` $\langle comma list \rangle$

Removes duplicate entries from the $\langle comma list \rangle$, leaving left most entry in the $\langle comma list \rangle$. The removal is applied globally.

`\clist_remove_element:Nn` `\clist_remove_element:Nn` $\langle comma list \rangle$ $\{ \langle entry \rangle \}$

Removes each occurrence of $\langle entry \rangle$ from the $\langle comma list \rangle$, where $\langle entry \rangle$ cannot contain the tokens $\{, \}$ or $\#$ (assuming normal \TeX category codes). The removal is local to the current \TeX group.

`\clist_gremove_element:Nn` `\clist_gremove_element:Nn` $\langle comma list \rangle$ $\{ \langle entry \rangle \}$

Removes each occurrence of $\langle entry \rangle$ from the $\langle comma list \rangle$, where $\langle entry \rangle$ cannot contain the tokens $\{, \}$ or $\#$ (assuming normal \TeX category codes). The removal applied globally.

64 Functions for ‘comma-list stacks’

Special comma-lists in $\text{\LaTeX}3$ are ‘stacks’ with their usual operations of ‘push’, ‘pop’, and ‘top’. They are internally implemented as comma-lists and share some of the functions (like `\clist_new:N` etc.)

`\clist_push:Nn`
`\clist_push:NV`
`\clist_push:No`
`\clist_push:cn`
`\clist_gpush:Nn`
`\clist_gpush:NV`
`\clist_gpush:No`
`\clist_gpush:cn` `\clist_push:Nn` $\langle stack \rangle$ $\{ \langle token list \rangle \}$

Locally or globally pushes $\langle token list \rangle$ as a single item onto the $\langle stack \rangle$. $\langle token list \rangle$ might

get expanded before the operation.

<code>\clist_pop:NN</code>
<code>\clist_pop:cN</code>
<code>\clist_gpop:NN</code>
<code>\clist_gpop:cN</code>

`\clist_pop:NN` $\langle stack \rangle$ $\langle tl\ var. \rangle$

Functions that assign the top item of $\langle stack \rangle$ to the token list variable $\langle tl\ var. \rangle$ and removes it from $\langle stack \rangle$!

<code>\clist_top:NN</code>
<code>\clist_top:cN</code>

`\clist_top:NN` $\langle stack \rangle$ $\langle tl\ var. \rangle$

Functions that locally assign the top item of $\langle stack \rangle$ to the token list variable $\langle tl\ var. \rangle$. Item is not removed from $\langle stack \rangle$!

65 Internal functions

<code>\clist_if_empty_err:N</code>

`\clist_if_empty_err:N` $\langle comma-list \rangle$

Signals an L^AT_EX3 error if $\langle comma-list \rangle$ is empty.

<code>\clist_pop_aux:nnNN</code>

`\clist_pop_aux:nnNN` $\langle assign_1 \rangle$ $\langle assign_2 \rangle$ $\langle comma-list \rangle$ $\langle tl\ var. \rangle$

Function that assigns the left-most item of $\langle comma-list \rangle$ to $\langle tl\ var. \rangle$ using $\langle assign_1 \rangle$ and assigns the tail to $\langle comma-list \rangle$ using $\langle assign_2 \rangle$. This function could be used to implement a global return function.

<code>\clist_get_aux:w</code>
<code>\clist_pop_aux:w</code>
<code>\clist_pop_auxi:w</code>
<code>\clist_put_aux:NNnnNn</code>

Functions used to implement put and get operations. They are not for meant for direct use.

Part XIV

The l3prop package

Property Lists

L^AT_EX3 implements a data structure called a ‘property list’ which allows arbitrary information to be stored and accessed using keywords rather than numerical indexing.

A property list might contain a set of keys such as **name**, **age**, and **ID**, which each have individual values that can be saved and retrieved.

66 Functions

<code>\prop_new:N</code>
<code>\prop_new:c</code>

`\prop_new:N <prop>`
 Defines *<prop>* to be a variable of type *<prop>*.

<code>\prop_clear:N</code>
<code>\prop_clear:c</code>
<code>\prop_gclear:N</code>
<code>\prop_gclear:c</code>

`\prop_clear:N <prop>`
 These functions locally or globally clear *<prop>*.

<code>\prop_put:Nnn</code>
<code>\prop_put:Nno</code>
<code>\prop_put:NnV</code>
<code>\prop_put:NVn</code>
<code>\prop_put:NVV</code>
<code>\prop_put:Nnx</code>
<code>\prop_put:cnn</code>
<code>\prop_put:cnx</code>
<code>\prop_gput:Nnn</code>
<code>\prop_gput:NVn</code>
<code>\prop_gput:Nno</code>
<code>\prop_gput:NnV</code>
<code>\prop_gput:Nnx</code>
<code>\prop_gput:cnn</code>
<code>\prop_gput:ccx</code>

`\prop_put:Nnn <prop> {<key>} {<token list>}`
 Locally or globally associates *<token list>* with *<key>* in the *<prop>* *<prop>*. If *<key>* has already a meaning within *<prop>* this value is overwritten.
 The *<key>* must not contain unescaped # tokens but the *<token list>* may.

<code>\prop_gput_if_new:Nnn</code>

`\prop_gput_if_new:Nnn <prop> {<key>} {<token list>}`

Globally associates $\langle token\ list \rangle$ with $\langle key \rangle$ in the $\langle prop \rangle$ $\langle prop \rangle$ but only if $\langle key \rangle$ has so far no meaning within $\langle prop \rangle$. Silently ignored if $\langle key \rangle$ is already set in the $\langle prop \rangle$.

<code>\prop_get:NnN</code>
<code>\prop_get:NVN</code>
<code>\prop_get:cnN</code>
<code>\prop_get:cVN</code>
<code>\prop_gget:NnN</code>
<code>\prop_gget:NVN</code>
<code>\prop_gget:cnN</code>
<code>\prop_gget:cVN</code>

`\prop_get:NnN $\langle prop \rangle$ { $\langle key \rangle$ } $\langle tl\ var. \rangle$`

If $\langle info \rangle$ is the information associated with $\langle key \rangle$ in the $\langle prop \rangle$ $\langle prop \rangle$ then the token list variable $\langle tl\ var. \rangle$ gets $\langle info \rangle$ assigned. Otherwise its value is the special quark `\q_no_value`. The assignment is done either locally or globally.

<code>\prop_set_eq:NN</code>
<code>\prop_set_eq:cN</code>
<code>\prop_set_eq:Nc</code>
<code>\prop_set_eq:cc</code>
<code>\prop_gset_eq:NN</code>
<code>\prop_gset_eq:cN</code>
<code>\prop_gset_eq:Nc</code>
<code>\prop_gset_eq:cc</code>

`\prop_set_eq:NN $\langle prop_1 \rangle$ $\langle prop_2 \rangle$`

A fast assignment of $\langle prop \rangle$ s.

<code>\prop_get_gdel:NnN</code>

`\prop_get_gdel:NnN $\langle prop \rangle$ { $\langle key \rangle$ } $\langle tl\ var. \rangle$`

Like `\prop_get:NnN` but additionally removes $\langle key \rangle$ (and its $\langle info \rangle$) from $\langle prop \rangle$.

<code>\prop_del:Nn</code>
<code>\prop_del:NV</code>
<code>\prop_gdel:Nn</code>
<code>\prop_gdel:NV</code>

`\prop_del:Nn $\langle prop \rangle$ { $\langle key \rangle$ }`

Locally or globally deletes $\langle key \rangle$ and its $\langle info \rangle$ from $\langle prop \rangle$ if found. Otherwise does nothing.

<code>\prop_map_function:NN *</code>
<code>\prop_map_function:cN *</code>
<code>\prop_map_function:Nc *</code>
<code>\prop_map_function:cc *</code>

`\prop_map_function:NN $\langle prop \rangle$ $\langle function \rangle$`

Maps $\langle function \rangle$ which should be a function with two arguments ($\langle key \rangle$ and $\langle info \rangle$) over every $\langle key \rangle$ $\langle info \rangle$ pair of $\langle prop \rangle$. Property lists do not have any intrinsic “order” when

stored. As a result, you should not expect any particular order to apply when using these mapping functions, even with newly-created properly lists.

<code>\prop_map_inline:Nn</code>
<code>\prop_map_inline:cn</code>

`\prop_map_inline:Nn <prop> {<inline function>}`

Just like `\prop_map_function:NN` but with the function of two arguments supplied as inline code. Within `<inline function>` refer to the arguments via `#1 (<key>)` and `#2 (<info>)`. Nestable. Property lists do not have any intrinsic “order” when stored. As a result, you should not expect any particular order to apply when using these mapping functions, even with newly-created properly lists.

<code>\prop_map_break:</code>

`\prop_map_inline:Nn <prop> {
... \<break test>:T {\prop_map_break:} }`

For breaking out of a loop. To be used inside TF-type functions as shown in the example above.

<code>\prop_show:N</code>
<code>\prop_show:c</code>

`\prop_show:N <prop>`

Pauses the compilation and shows `<prop>` on the terminal output and in the log file.

<code>\prop_display:N</code>
<code>\prop_display:c</code>

`\prop_display:N <prop>`

As with `\prop_show:N` but pretty prints the output one line per property pair.

67 Predicates and conditionals

<code>\prop_if_empty_p:N</code>
<code>\prop_if_empty_p:c</code>

`\prop_if_empty_p:N <prop> {\<true code>} {\<false code>}`

Predicates to test whether or not a particular `<prop>` is empty.

<code>\prop_if_empty:N\underline{TF} *</code>
<code>\prop_if_empty:c\underline{TF} *</code>

`\prop_if_empty:N \underline{TF} <prop> {\<true code>} {\<false code>}`

Set of conditionals that test whether or not a particular $\langle prop \rangle$ is empty.

<code>\prop_if_eq_p:NN</code>	★	
<code>\prop_if_eq_p:cN</code>	★	
<code>\prop_if_eq_p:Nc</code>	★	
<code>\prop_if_eq_p:cc</code>	★	
<code>\prop_if_eq:NN\overline{TF}</code>	★	
<code>\prop_if_eq:cN\overline{TF}</code>	★	
<code>\prop_if_eq:Nc\overline{TF}</code>	★	
<code>\prop_if_eq:cc\overline{TF}</code>	★	
<code>\prop_if_eq:NnF $\langle prop_1 \rangle$ $\langle prop_2 \rangle$ {$\langle false code \rangle$}</code>		
Execute $\langle false code \rangle$ if $\langle prop_1 \rangle$ doesn't hold the same token list as $\langle prop_2 \rangle$. Only expandable for new versions of pdfTeX.		

<code>\prop_if_in:Nn\overline{TF}</code>	
<code>\prop_if_in:NV\overline{TF}</code>	
<code>\prop_if_in:No\overline{TF}</code>	
<code>\prop_if_in:cn\overline{TF}</code>	
<code>\prop_if_in:cc\overline{TF}</code>	
<code>\prop_if_in:NnTF $\langle prop \rangle$ {$\langle key \rangle$} {$\langle true code \rangle$} {$\langle false code \rangle$}</code>	
Tests if $\langle key \rangle$ is used in $\langle prop \rangle$ and then either executes $\langle true code \rangle$ or $\langle false code \rangle$.	

68 Internal functions

<code>\q_prop</code>	Quark used to delimit property lists internally.
----------------------	--

<code>\prop_put_aux:w</code>	
<code>\prop_put_if_new_aux:w</code>	Internal functions implementing the put operations.

<code>\prop_get_aux:w</code>	
<code>\prop_gget_aux:w</code>	
<code>\prop_get_del_aux:w</code>	
<code>\prop_del_aux:w</code>	Internal functions implementing the get and delete operations.

<code>\prop_if_in_aux:w</code>	Internal function implementing the key test operation.
--------------------------------	--

<code>\prop_map_function_aux:w</code>	Internal function implementing the map operations.
---------------------------------------	--

<code>\g_prop_inline_level_int</code>	Integer used in internal name for function used inside
<code>\prop_map_inline:NN</code>	

<code>\prop_split_aux:Nnn</code>	<code>\prop_split_aux:Nnn $\langle prop \rangle$ $\langle key \rangle$ $\langle cmd \rangle$</code>
Internal function that invokes $\langle cmd \rangle$ with 3 arguments: 1st is the beginning of $\langle prop \rangle$	

before $\langle key \rangle$, 2nd is the value associated with $\langle key \rangle$, 3rd is the rest of $\langle prop \rangle$ after $\langle key \rangle$. If there is no key $\langle key \rangle$ in $\langle prop \rangle$, then the 2 arg is `\q_no_value` and the 3rd arg is empty; otherwise the 3rd argument has the two extra tokens $\langle key \rangle$ `\q_no_value` at the end.

This function is used to implement various get operations.

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_`.

69 Generic functions

<code>\box_new:N</code>	<code>\box_new:N</code> $\langle box \rangle$
<code>\box_new:c</code>	

Defines $\langle box \rangle$ to be a new variable of type `box`.

T_EXhackers note: `\box_new:N` is the equivalent of plain T_EX's `\newbox`.

<code>\if_hbox:N</code>	<code>\if_hbox:N</code> $\langle box \rangle$ $\langle true\ code \rangle$ <code>\else:</code> $\langle false\ code \rangle$ <code>\fi:</code>
<code>\if_vbox:N</code>	
<code>\if_box_empty:N</code>	

`\if_hbox:N` and `\if_vbox:N` check if $\langle box \rangle$ is an horizontal or vertical box resp. `\if_box_empty:N` tests if $\langle box \rangle$ is empty (void) and executes `code` according to the test outcome.

T_EXhackers note: These are the T_EX primitives `\ifhbox`, `\ifvbox` and `\ifvoid`.

<code>\box_if_horizontal_p:N</code>	<code>\box_if_horizontal:N</code> $\langle box \rangle$ $\{ \langle true\ code \rangle \}$ $\{ \langle false\ code \rangle \}$
<code>\box_if_horizontal_p:c</code>	
<code>\box_if_horizontal:N\overline{TF}</code>	
<code>\box_if_horizontal:c\overline{TF}</code>	

Tests if $\langle box \rangle$ is an horizontal box and executes $\langle code \rangle$ accordingly.

<code>\box_if_vertical_p:N</code> <code>\box_if_vertical_p:c</code> <code>\box_if_vertical:NTF</code> <code>\box_if_vertical:cTF</code>	<code>\box_if_vertical:NTF</code> $\langle box \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
--	--

Tests if $\langle box \rangle$ is a vertical box and executes $\langle code \rangle$ accordingly.

<code>\box_if_empty_p:N</code> <code>\box_if_empty_p:c</code> <code>\box_if_empty:NTF</code> <code>\box_if_empty:cTF</code>	<code>\box_if_empty:NTF</code> $\langle box \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$
--	---

Tests if $\langle box \rangle$ is empty (void) and executes `code` according to the test outcome.

T_EXhackers note: `\box_if_empty:NTF` is the L^AT_EX3 function name for `\ifvoid`.

<code>\box_set_eq:NN</code> <code>\box_set_eq:cN</code> <code>\box_set_eq:Nc</code> <code>\box_set_eq:cc</code> <code>\box_set_eq_clear:NN</code> <code>\box_set_eq_clear:cN</code> <code>\box_set_eq_clear:Nc</code> <code>\box_set_eq_clear:cc</code>	<code>\box_set_eq:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
--	---

Sets $\langle box_1 \rangle$ equal to $\langle box_2 \rangle$. The `_clear` versions eradicate the contents of $\langle box_2 \rangle$ afterwards.

<code>\box_gset_eq:NN</code> <code>\box_gset_eq:cN</code> <code>\box_gset_eq:Nc</code> <code>\box_gset_eq:cc</code> <code>\box_gset_eq_clear:NN</code> <code>\box_gset_eq_clear:cN</code> <code>\box_gset_eq_clear:Nc</code> <code>\box_gset_eq_clear:cc</code>	<code>\box_gset_eq:NN</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$
--	--

Globally sets $\langle box_1 \rangle$ equal to $\langle box_2 \rangle$. The `_clear` versions eradicate the contents of

$\langle box_2 \rangle$ afterwards.

<code>\box_set_to_last:N</code>
<code>\box_set_to_last:c</code>
<code>\box_gset_to_last:N</code>
<code>\box_gset_to_last:c</code>

`\box_set_to_last:N $\langle box \rangle$`

Sets $\langle box \rangle$ equal to the previous box `\l_last_box` and removes `\l_last_box` from the current list (unless in outer vertical or math mode).

<code>\box_move_right:nn</code>
<code>\box_move_left:nn</code>
<code>\box_move_up:nn</code>
<code>\box_move_down:nn</code>

`\box_move_left:nn { $\langle dimen \rangle$ } { $\langle box function \rangle$ }`

Moves $\langle box function \rangle$ $\langle dimen \rangle$ in the direction specified. $\langle box function \rangle$ is either an operation on a box such as `\box_use:N` or a “raw” box specification like `\vbox:n{xyz}`.

<code>\box_clear:N</code>
<code>\box_clear:c</code>
<code>\box_gclear:N</code>
<code>\box_gclear:c</code>

`\box_clear:N $\langle box \rangle$`

Clears $\langle box \rangle$ by setting it to the constant `\c_void_box`. `\box_gclear:N` does it globally.

<code>\box_use:N</code>
<code>\box_use:c</code>
<code>\box_use_clear:N</code>
<code>\box_use_clear:c</code>

`\box_use:N $\langle box \rangle$`
`\box_use_clear:N $\langle box \rangle$`

`\box_use:N` puts a copy of $\langle box \rangle$ on the current list while `\box_use_clear:N` puts the box on the current list and then eradicates the contents of it.

T_EXhackers note: `\box_use:N` and `\box_use_clear:N` are the T_EX primitives `\copy` and `\box` with new (descriptive) names.

<code>\box_ht:N</code>
<code>\box_ht:c</code>
<code>\box_dp:N</code>
<code>\box_dp:c</code>
<code>\box_wd:N</code>
<code>\box_wd:c</code>

`\box_ht:N $\langle box \rangle$`

Returns the height, depth, and width of $\langle box \rangle$ for use in dimension settings.

T_EXhackers note: These are the T_EX primitives `\ht`, `\dp` and `\wd`.

<code>\box_set_dp:Nn</code> <code>\box_set_dp:cn</code>
--

`\box_set_dp:Nn` $\langle box \rangle$ $\{ \langle dimension\ expression \rangle \}$

Set the depth(below the baseline) of the $\langle box \rangle$ to the value of the $\{ \langle dimension\ expression \rangle \}$. This is a local assignment.

<code>\box_set_ht:Nn</code> <code>\box_set_ht:cn</code>
--

`\box_set_ht:Nn` $\langle box \rangle$ $\{ \langle dimension\ expression \rangle \}$

Set the height(above the baseline) of the $\langle box \rangle$ to the value of the $\{ \langle dimension\ expression \rangle \}$. This is a local assignment.

<code>\box_set_wd:Nn</code> <code>\box_set_wd:cn</code>
--

`\box_set_wd:Nn` $\langle box \rangle$ $\{ \langle dimension\ expression \rangle \}$

Set the width of the $\langle box \rangle$ to the value of the $\{ \langle dimension\ expression \rangle \}$. This is a local assignment.

<code>\box_show:N</code> <code>\box_show:c</code>
--

`\box_show:N` $\langle box \rangle$

Writes the contents of $\langle box \rangle$ to the log file.

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

<code>\c_empty_box</code> <code>\l_tmpa_box</code> <code>\l_tmpb_box</code>

`\c_empty_box` is the constantly empty box. The others are scratch boxes.

<code>\l_last_box</code>

`\l_last_box` is more or less a read-only box register managed by the engine. It denotes the last box on the current list if there is one, otherwise it is void. You can set other boxes to this box, with the result that the last box on the current list is removed at the same time (so it is with variable with side-effects).

70 Horizontal mode

<code>\hbox:n</code>	<code>\hbox:n {<contents>}</code>
----------------------	---

Places a `hbox` of natural size.

<code>\hbox_set:Nn</code>	<code>\hbox_set:Nn <box> {<contents>}</code>
<code>\hbox_set:cn</code>	
<code>\hbox_gset:Nn</code>	
<code>\hbox_gset:cn</code>	

Sets `<box>` to be a horizontal mode box containing `<contents>`. It has its natural size. `\hbox_gset:Nn` does it globally.

<code>\hbox_set_to_wd:Nnn</code>	<code>\hbox_set_to_wd:Nnn <box> {<dimen>} {<contents>}</code>
<code>\hbox_set_to_wd:cnn</code>	
<code>\hbox_gset_to_wd:Nnn</code>	
<code>\hbox_gset_to_wd:cnn</code>	

Sets `<box>` to contain `<contents>` and have width `<dimen>`. `\hbox_gset_to_wd:Nn` does it globally.

<code>\hbox_to_wd:nn</code>	<code>\hbox_to_wd:nn {<dimen>} <contents></code>
<code>\hbox_to_zero:n</code>	

Places a `<box>` of width `<dimen>` containing `<contents>`. `\hbox_to_zero:n` is a shorthand for a width of zero.

<code>\hbox_overlap_left:n</code>	<code>\hbox_overlap_left:n <contents></code>
<code>\hbox_overlap_right:n</code>	

Places a `<box>` of width zero containing `<contents>` in a way that it overlaps with surrounding material (sticking out to the left or right).

<code>\hbox_set_inline_begin:N</code>	<code>\hbox_set_inline_begin:N <box> <contents></code>
<code>\hbox_set_inline_begin:c</code>	
<code>\hbox_set_inline_end:</code>	
<code>\hbox_gset_inline_begin:N</code>	
<code>\hbox_gset_inline_begin:c</code>	
<code>\hbox_gset_inline_end:</code>	

Sets $\langle box \rangle$ to contain $\langle contents \rangle$. This type is useful for use in environment definitions.

$\backslash hbox_unpack:N$ $\backslash hbox_unpack:c$ $\backslash hbox_unpack_clear:N$ $\backslash hbox_unpack_clear:c$	$\backslash hbox_unpack:N \langle box \rangle$
--	---

$\backslash hbox_unpack:N$ unpacks the contents of the $\langle box \rangle$ register and $\backslash hbox_unpack_clear:N$ also clears the $\langle box \rangle$ after unpacking it.

T_EXhackers note: These are the T_EX primitives $\backslash unhcopy$ and $\backslash unhbox$.

71 Vertical mode

$\backslash vbox:n$	$\backslash vbox:n \{ \langle contents \rangle \}$
---------------------	--

Places a $vbox$ of natural size with baseline equal to the baseline of the last object in the box, i.e., if the last object is a line of text the box has the same depth as that line; otherwise the depth will be zero.

$\backslash vbox_top:n$	$\backslash vbox_top:n \{ \langle contents \rangle \}$
--------------------------	---

Same as $\backslash vbox:n$ except that the reference point will be at the baseline of the first object in the box not the last.

$\backslash vbox_set:Nn$ $\backslash vbox_set:cn$ $\backslash vbox_gset:Nn$ $\backslash vbox_gset:cn$	$\backslash vbox_set:Nn \langle box \rangle \{ \langle contents \rangle \}$
--	--

Sets $\langle box \rangle$ to be a vertical mode box containing $\langle contents \rangle$. It has its natural size and the reference point will be at the baseline of the last object in the box. $\backslash vbox_gset:Nn$ does it globally.

$\backslash vbox_set_top:Nn$ $\backslash vbox_set_top:cn$ $\backslash vbox_gset_top:Nn$ $\backslash vbox_gset_top:cn$	$\backslash vbox_set_top:Nn \langle box \rangle \{ \langle contents \rangle \}$
--	---

Sets $\langle box \rangle$ to be a vertical mode box containing $\langle contents \rangle$. It has its natural size (usually

a small height and a larger depth) and the reference point will be at the baseline of the first object in the box. `\vbox_gset_top:Nn` does it globally.

<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> <code>\vbox_gset_to_ht:ccn</code>	<code>\vbox_set_to_ht:Nnn</code> $\langle box \rangle$ $\{\langle dimen \rangle\}$ $\{\langle contents \rangle\}$
---	---

Sets $\langle box \rangle$ to contain $\langle contents \rangle$ and have total height $\langle dimen \rangle$. `\vbox_gset_to_ht:Nn` does it globally.

<code>\vbox_set_inline_begin:N</code> <code>\vbox_set_inline_end:</code> <code>\vbox_gset_inline_begin:N</code> <code>\vbox_gset_inline_end:</code>	<code>\vbox_set_inline_begin:N</code> $\langle box \rangle$ $\langle contents \rangle$ <code>\vbox_set_inline_end:</code>
--	--

Sets $\langle box \rangle$ to contain $\langle contents \rangle$. This type is useful for use in environment definitions.

<code>\vbox_set_split_to_ht:NNn</code>	<code>\vbox_set_split_to_ht:NNn</code> $\langle box_1 \rangle$ $\langle box_2 \rangle$ $\{\langle dimen \rangle\}$
--	--

Sets $\langle box_1 \rangle$ to contain the top $\langle dimen \rangle$ part of $\langle box_2 \rangle$.

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

<code>\vbox_to_ht:nn</code> <code>\vbox_to_zero:n</code>	<code>\vbox_to_ht:nn</code> $\{\langle dimen \rangle\}$ $\langle contents \rangle$ <code>\vbox_to_zero:n</code> $\langle contents \rangle$
---	---

Places a $\langle box \rangle$ of size $\langle dimen \rangle$ containing $\langle contents \rangle$.

<code>\vbox_unpack:N</code> <code>\vbox_unpack:c</code> <code>\vbox_unpack_clear:N</code> <code>\vbox_unpack_clear:c</code>	<code>\vbox_unpack:N</code> $\langle box \rangle$
--	---

`\vbox_unpack:N` unpacks the contents of the $\langle box \rangle$ register and `\vbox_unpack_clear:N` also clears the $\langle box \rangle$ after unpacking it.

T_EXhackers note: These are the T_EX primitives `\unvcopy` and `\unvbox`.

Part XVI

The l3io package

Low-level file i/o

Reading and writing from file streams is handled in L^AT_EX3 using functions with prefixes `\iow_...` (file reading) and `\ior_...` (file writing). Many of the basic functions are very similar, with reading and writing using the same syntax and function concepts. As a result, the reading and writing functions are documented together where this makes sense.

As T_EX is limited to 16 input streams and 16 output streams, direct use of the streams by the programmer is not supported in L^AT_EX3. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Reading from or writing to a file requires a $\langle stream \rangle$ to be used. This is a csname which refers to the file being processed, and is independent of the name of the file (except of course that the file name is needed when the file is opened).

72 Opening and closing streams

<code>\iow_new:N</code>
<code>\iow_new:c</code>
<code>\ior_new:N</code>
<code>\ior_new:c</code>

`\iow_new:N $\langle stream \rangle$`

Reserves the name $\langle stream \rangle$ for use in accessing a file stream. This operation does not open a raw T_EX stream, which is handled internally using a pool and is should not be accessed directly by the programmer.

<code>\iow_open:Nn</code>
<code>\iow_open:cn</code>
<code>\ior_open:Nn</code>
<code>\ior_open:cn</code>

`\iow_open:Nn $\langle stream \rangle$ { $\langle file name \rangle$ }`
`\iow_open:Nn $\langle stream \rangle$ { $\langle file name \rangle$ }`

Opens $\langle file name \rangle$ for writing (`\iow_...`) or reading (`\ior_...`) using $\langle stream \rangle$ as the csname by which the file is accessed. If $\langle stream \rangle$ was already open (for either writing or reading) it is closed before the new operation begins. The $\langle stream \rangle$ is available for access immediately after issuing an `open` instruction. The $\langle stream \rangle$ will remain allocated to $\langle file name \rangle$ until a `close` instruction is given or at the end of the T_EX run.

Opening a file for writing will clear any existing content in the file (*i.e.* writing is *not* additive). As the total number of writing streams is limited, it may well be best to save material to be written to an intermediate storage format (for example a token list or toks), and to write the material in one ‘shot’ from this variable. In this way the file stream is only required for a limited time.

<code>\iow_close:N</code>	
<code>\iow_close:c</code>	
<code>\ior_close:N</code>	<code>\iow_close:N <stream></code>
<code>\ior_close:c</code>	<code>\ior_close:N <stream></code>

Closes $\langle stream \rangle$, freeing up one of the underlying \TeX streams for reuse. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers (the resources here are limited). The name of the $\langle stream \rangle$ will be freed at this stage, to ensure that any further attempts to write to it result in an error.

<code>\iow_open_streams:</code>	
<code>\iow_open_streams:</code>	<code>\iow_open_streams:</code>

Displays a list of the file names associated with each open stream: intended for tracking down problems.

72.1 Writing to files

<code>\iow_now:Nx</code>	
<code>\iow_now:Nn</code>	<code>\iow_now:Nx <stream> {\tokens}</code>

`\iow_now:Nx` immediately writes the expansion of $\langle tokens \rangle$ to the output $\langle stream \rangle$. If the $\langle stream \rangle$ is not open output goes to the terminal. The variant `\iow_now:Nn` writes out $\langle tokens \rangle$ without any further expansion.

\TeX hackers note: These are the equivalent of \TeX ’s `\immediate\write` with and without expansion control.

<code>\iow_log:n</code>	
<code>\iow_log:x</code>	
<code>\iow_term:n</code>	
<code>\iow_term:x</code>	<code>\iow_log:x {\tokens}</code>

These are dedicated functions which write to the log (transcript) file and the terminal, respectively. They are equivalent to using `\iow_now:N(n/x)` to the streams `\c_iow_log_stream` and `\c_iow_term_stream`. The writing takes place immediately.

<code>\iow_now_buffer_safe:Nn</code>	
<code>\iow_now_buffer_safe:Nx</code>	<code>\iow_now_buffer_safe:Nn <stream> {\tokens}</code>

Immediately write $\langle tokens \rangle$ expanded to $\langle stream \rangle$, with every space converted into a newline. This means that the file can be read back without the danger that very long lines overflow T_EX's buffer.

$\backslash iow_now_when_avail:Nn$ $\backslash iow_now_when_avail:cn$ $\backslash iow_now_when_avail:Nx$ $\backslash iow_now_when_avail:Cx$	$\backslash iow_now_when_avail:Nn \langle stream \rangle \{ \langle tokens \rangle \}$
--	---

If $\langle stream \rangle$ is open, writes the $\langle tokens \rangle$ to the $\langle stream \rangle$ in the same manner as $\backslash iow_now:N(n/x)$. If the $\langle stream \rangle$ is not open, the $\langle tokens \rangle$ are simply thrown away.

$\backslash iow_shipout:Nx$ $\backslash iow_shipout:Nn$	$\backslash iow_shipout:Nx \langle stream \rangle \{ \langle tokens \rangle \}$
--	--

Write $\langle tokens \rangle$ to $\langle stream \rangle$ at the point at which the current page is finished. The $\langle tokens \rangle$ are either written unexpanded ($\backslash iow_shipout:Nn$) or expanded only at the point that the function is used ($\backslash iow_shipout:Nx$), *i.e.* no expansion takes place when writing to the file.

$\backslash iow_shipout_x:Nx$ $\backslash iow_shipout_x:Nn$	$\backslash iow_shipout_x:Nx \langle stream \rangle \{ \langle tokens \rangle \}$
--	---

Write $\langle tokens \rangle$ to $\langle stream \rangle$ at the point at which the current page is finished. The $\langle tokens \rangle$ are expanded at the time of writing in addition to any expansion at the time of use of the function. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

T_EXhackers note: These are the equivalent of T_EX's $\backslash write$ with and without expansion control at point of use.

$\backslash iow_newline: *$	$\backslash iow_newline:$
------------------------------	----------------------------

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 a $\backslash iow_now:Nn$ call).

$\backslash iow_char:N *$	$\backslash iow_char:N \backslash \langle char \rangle$ $\backslash iow_char:N \%$
----------------------------	---

Inserts $\langle char \rangle$ into the output stream. Useful when trying to write difficult characters such as %, {, }, *etc.* in messages, for example:

```
 $\backslash iow\_now:Nx \backslash g\_my\_stream \{ \backslash iow\_char:N \{ text \backslash iow\_char:N \} \}$ 
```

The function has no effect if writing is taking place without expansion (*e.g.* in a $\backslash iow_now:Nn$ call).

72.2 Reading from files

<code>\ior_to:NN</code> <code>\ior_gto:NN</code>

`\ior_to:NN <stream> <token list variable>`

Functions that reads one or more lines (until an equal number of left and right braces are found) from the input stream *<stream>* and places the result locally or globally into the *<token list variable>*. If *<stream>* is not open, input is requested from the terminal.

<code>\ior_if_eof_p:N *</code> <code>\ior_if_eof:NTF *</code>
--

`\ior_if_eof:NTF <stream> {<true code>} {<false code>}`

Tests if the end of a *<stream>* has been reached during a reading operation. The test will also return a `true` value if the *<stream>* is not open or the *<file name>* associated with a *<stream>* does not exist at all.

73 Internal functions

<code>\iow_raw_new:N</code> <code>\iow_raw_new:c</code> <code>\ior_raw_new:N</code> <code>\ior_raw_new:c</code>
--

`\iow_raw_new:N <stream>`

Creates a new low-level *<stream>* for use in subsequent functions. As allocations are made using a pool *do not use this function!*

T_EXhackers note: This is L^AT_EX 2_ε's `\newwrite`.

<code>\if_eof:w *</code>

`\if_eof:w <stream> <true code> \else: <false code> \fi:`

Tests if the end of *<stream>* has been reached during a reading operation.

T_EXhackers note: This is the primitive `\ifeof`.

74 Variables and constants

<code>\c_io_streams_tl</code>

A list of the positions available for stream allocation (numbers 0

to 15).

<code>\c_iow_term_stream</code>
<code>\c_ior_term_stream</code>
<code>\c_iow_log_stream</code>
<code>\c_ior_log_stream</code>

Fixed stream numbers for accessing to the log and the terminal. The reading and writing values are the same but are provided so that the meaning is clear.

<code>\g_iow_streams_prop</code>
<code>\g_ior_streams_prop</code>

Allocation records for streams, linking the stream number to the current name being used for that stream.

<code>\g_iow_tmp_stream</code>
<code>\g_ior_tmp_stream</code>

Used when creating new streams at the \TeX level.

<code>\l_iow_stream_int</code>
<code>\l_ior_stream_int</code>

Number of stream currently being allocated.

Part XVII

The **l3msg** package

Communicating with the user

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 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.

75 Creating new messages

All messages have to be created before they can be used. Inside the message text, spaces are *not* ignored. A space where T_EX would normally gobble one can be created using `\`, and a new line with `\\`. New lines may have ‘continuation’ text added by the output system.

<code>\msg_new:nnnn</code> <code>\msg_new:nnn</code> <code>\msg_set:nnnn</code> <code>\msg_set:nnn</code>	<code>\msg_new:nnnn {<module>} {<message>} {<text>}</code> <code>{<more text>}</code>
--	--

Creates new `<message>` for `<module>` to produce `<text>` initially and `<more text>` if requested by the user. `<text>` and `<more text>` can use up to four macro parameters (**#1** to **#4**), which are supplied by the message system. At the point where `<message>` is printed, the material supplied for **#1** to **#4** will be subject to an x-type expansion.

An error will be raised by the **new** functions if the message already exists: the **set** functions do not carry any checking. For messages defined using `\msg_new:nnn` or `\msg_set:nnn` L^AT_EX3 will supply a standard `<more text>` at the point the message is used, if this is required.

76 Message classes

Creating message output requires the message to be given a class.

<code>\msg_class_new:nn</code> <code>\msg_class_set:nn</code>	<code>\msg_class_new:nn {<class>} {<code>}</code>
--	---

Creates new `<class>` to output a message, using `<code>` to process the message text. The `<class>` should be a text value, while the `<code>` may be any arbitrary material.

The module defines several common message classes. The following describes the standard behaviour of each class if no redirection of the class or message is active. In all cases, the message may be issued supplying 0 to 4 arguments. The code will ensure that there are no errors if the number of arguments supplied here does not match the number in the definition of the message (although of course the sense of the message may be impaired).

<code>\msg_fatal:nnxxxx</code> <code>\msg_fatal:nnxxx</code> <code>\msg_fatal:nnxx</code> <code>\msg_fatal:nnx</code> <code>\msg_fatal:nn</code>	<code>\msg_fatal:nnxxxx {<module>} {<name>} {<arg one>}</code> <code>{<arg two>} {<arg three>} {<arg four>}</code>
--	---

Issues `<module>` error message `<name>`, passing `<arg one>` to `<arg four>` to the text-creating

functions. After issuing a fatal error the T_EX run will halt.

<pre> \msg_error:nnxxxx \msg_error:nnxxx \msg_error:nnxx \msg_error:nnx \msg_error:nn </pre>	<pre> \msg_error:nnxxxx {<module>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>} </pre>
--	---

Issues *<module>* error message *<name>*, passing *<arg one>* to *<arg four>* to the text-creating functions.

T_EXhackers note: The standard output here is similar to `\PackageError`.

<pre> \msg_warning:nnxxxx \msg_warning:nnxxx \msg_warning:nnxx \msg_warning:nnx \msg_warning:nn </pre>	<pre> \msg_warning:nnxxxx {<module>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>} </pre>
--	---

Prints *<module>* message *<name>* to the terminal, passing *<arg one>* to *<arg four>* to the text-creating functions.

T_EXhackers note: The standard output here is similar to `\PackageWarningNoLine`.

<pre> \msg_info:nnxxxx \msg_info:nnxxx \msg_info:nnxx \msg_info:nnx \msg_info:nn </pre>	<pre> \msg_info:nnxxxx {<module>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>} </pre>
---	--

Prints *<module>* message *<name>* to the log, passing *<arg one>* to *<arg four>* to the text-creating functions.

T_EXhackers note: The standard output here is similar to `\PackageInfoNoLine`.

<pre> \msg_log:nnxxxx \msg_log:nnxxx \msg_log:nnxx \msg_log:nnx \msg_log:nn </pre>	<pre> \msg_log:nnxxxx {<module>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>} </pre>
--	---

Prints *<module>* message *<name>* to the log, passing *<arg one>* to *<arg four>* to the text-

creating functions. No continuation text is added.

<pre>\msg_trace:nnxxxx \msg_trace:nnxxx \msg_trace:nnxx \msg_trace:nnx \msg_trace:nn</pre>	<pre>\msg_trace:nnxxxx {<module>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</pre>
--	---

Prints *<module>* message *<name>* to the log, passing *<arg one>* to *<arg four>* to the text-creating functions. No continuation text is added.

<pre>\msg_none:nnxxxx \msg_none:nnxxx \msg_none:nnxx \msg_none:nnx \msg_none:nn</pre>	<pre>\msg_none:nnxxxx {<module>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>}</pre>
---	--

Does nothing: used for redirecting other message classes. Gobbles arguments given.

77 Redirecting messages

<pre>\msg_redirect_class:nn</pre>	<pre>\msg_redirect_class:nn {<class one>} {<class two>}</pre>
-----------------------------------	---

Changes the behaviour of messages of *<class one>* so that they are processed using the code for those of *<class two>*. Multiple redirections are possible. Redirection to a missing class or infinite loops will raise errors when the messages are used, rather than at the point of redirection.

<pre>\msg_redirect_module:nnn</pre>	<pre>\msg_redirect_module:nnn {<module>} {<class one>} {<class two>}</pre>
-------------------------------------	--

Redirects message of *<class one>* for *<module>* to act as though they were from *<class two>*. Messages of *<class one>* from sources other than *<module>* are not affected by this redirection.

T_EXhackers note: This function can be used to make some messages ‘silent’ by default. For example, all of the `trace` messages of *<module>* could be turned off with:

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

<pre>\msg_redirect_name:nnn</pre>	<pre>\msg_redirect_name:nnn {<module>} {<message>} {<class>}</pre>
-----------------------------------	--

Redirects a specific $\langle message \rangle$ from a specific $\langle module \rangle$ to act as a member of $\langle class \rangle$ of messages.

TeXhackers note: This function can be used to make a selected message ‘silent’ without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

78 Support functions for output

<code>\msg_line_context:</code>	<code>\msg_line_context:</code>
---------------------------------	---------------------------------

Prints the text specified in `\c_msg_on_line_tl` followed by the current line in the current input file.

TeXhackers note: This is similar to the text added to messages by L^AT_EX 2_ε’s `\PackageWarning` and `\PackageInfo`.

<code>\msg_line_number:</code>	<code>\msg_line_number:</code>
--------------------------------	--------------------------------

Prints the current line number in the current input file.

<code>\msg_newline:</code> <code>\msg_two_newlines:</code>	<code>\msg_newline:</code>
---	----------------------------

Print one or two newlines with no continuation information.

79 Low-level functions

The low-level functions do not make assumptions about module names. The output functions here produce messages directly, and do not respond to redirection.

<code>\msg_generic_new:nnn</code> <code>\msg_generic_new:nn</code> <code>\msg_generic_set:nnn</code> <code>\msg_generic_set:nn</code>	<code>\msg_generic_new:nnn {$\langle name \rangle$} {$\langle text \rangle$} {$\langle more text \rangle$}</code>
--	--

Creates new message $\langle name \rangle$ to produce $\langle text \rangle$ initially and $\langle more text \rangle$ if requested by the user. $\langle text \rangle$ and $\langle more text \rangle$ can use up to four macro parameters (#1 to #4),

which are supplied by the message system. Inside $\langle text \rangle$ and $\langle more\ text \rangle$ spaces are not ignored.

$\backslash msg_direct_interrupt:xxxxx$	$\backslash msg_direct_interrupt:xxxxx\ \{\langle first\ line \rangle\}\ \{\langle text \rangle\}$ $\{\langle continuation \rangle\}\ \{\langle last\ line \rangle\}\ \{\langle more\ text \rangle\}$
---	--

Executes a \TeX error, interrupting compilation. The $\langle first\ line \rangle$ is displayed followed by $\langle text \rangle$ and the input prompt. $\langle more\ text \rangle$ is displays if requested by the user. If $\langle more\ text \rangle$ is blank a default is supplied. Each line of $\langle text \rangle$ (broken with $\backslash\backslash$) begins with $\langle continuation \rangle$ and finishes off with $\langle last\ line \rangle$. $\langle last\ line \rangle$ has a period appended to it; do not add one yourself.

$\backslash msg_direct_log:xx$ $\backslash msg_direct_term:xx$	$\backslash msg_direct_log:xx\ \{\langle text \rangle\}\ \{\langle continuation \rangle\}$
---	--

Prints $\langle text \rangle$ to either the log or terminal. New lines (broken with $\backslash\backslash$) start with $\langle continuation \rangle$.

80 Kernel-specific functions

$\backslash msg_kernel_new:nnnn$ $\backslash msg_kernel_new:nnn$ $\backslash msg_kernel_set:nnnn$ $\backslash msg_kernel_set:nnn$	$\backslash msg_kernel_new:nnnn\ \{\langle division \rangle\}\ \{\langle name \rangle\}\ \{\langle text \rangle\}$ $\{\langle more\ text \rangle\}$
--	--

Creates new kernel message $\langle name \rangle$ to produce $\langle text \rangle$ initially and $\langle more\ text \rangle$ if requested by the user. $\langle text \rangle$ and $\langle more\ text \rangle$ can use up to four macro parameters (#1 to #4), which are supplied by the message system. Kernel messages are divided into $\langle divisions \rangle$, roughly equivalent to the $\LaTeX 2_\epsilon$ package names used.

$\backslash msg_kernel_fatal:nnxxxx$ $\backslash msg_kernel_fatal:nnxxx$ $\backslash msg_kernel_fatal:nnxx$ $\backslash msg_kernel_fatal:nnx$ $\backslash msg_kernel_fatal:nn$	$\backslash msg_kernel_fatal:nnxx\ \{\langle division \rangle\}\ \{\langle name \rangle\}\ \{\langle arg\ one \rangle\}$ $\{\langle arg\ two \rangle\}\ \{\langle arg\ three \rangle\}\ \{\langle arg\ four \rangle\}$
--	---

Issues kernel error message $\langle name \rangle$ for $\langle division \rangle$, passing $\langle arg\ one \rangle$ to $\langle arg\ four \rangle$ to

the text-creating functions. The TeX run then halts. Cannot be redirected.

<pre> \msg_kernel_error:nnxxxx \msg_kernel_error:nnxxx \msg_kernel_error:nnxx \msg_kernel_error:nnx \msg_kernel_error:nn </pre>	<pre> \msg_kernel_error:nnxx {<division>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>} </pre>
---	--

Issues kernel error message $\langle name \rangle$ for $\langle division \rangle$, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions. Cannot be redirected.

<pre> \msg_kernel_warning:nnxxxx \msg_kernel_warning:nnxxx \msg_kernel_warning:nnxx \msg_kernel_warning:nnx \msg_kernel_warning:nn </pre>	<pre> \msg_kernel_warning:nnxx {<division>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>} </pre>
---	--

Prints kernel message $\langle name \rangle$ for $\langle division \rangle$ to the terminal, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions.

<pre> \msg_kernel_info:nnxxxx \msg_kernel_info:nnxxx \msg_kernel_info:nnxx \msg_kernel_info:nnx \msg_kernel_info:nn </pre>	<pre> \msg_kernel_info:nnx {<division>} {<name>} {<arg one>} {<arg two>} {<arg three>} {<arg four>} </pre>
--	--

Prints kernel message $\langle name \rangle$ for $\langle division \rangle$ to the log, passing $\langle arg one \rangle$ to $\langle arg four \rangle$ to the text-creating functions.

<pre> \msg_kernel_bug:x </pre>	<pre> \msg_kernel_bug:x {<text>} </pre>
--------------------------------	---

Short-cut for ‘This is a LaTeX bug: check coding’ errors.

<pre> \msg_fatal_text:n </pre>	<pre> \msg_fatal_text:n {<package>} </pre>
--------------------------------	--

Prints ‘Fatal $\langle package \rangle$ error’ for use in error messages.

81 Variables and constants

```
\c_msg_error_tl  
\c_msg_warning_tl  
\c_msg_info_tl
```

Simple headers for errors. Although these are marked as constants, they could be changed for printing errors in a different language.

```
\c_msg_coding_error_text_tl  
\c_msg_fatal_text_tl  
\c_msg_help_text_tl  
\c_msg_kernel_bug_text_tl  
\c_msg_kernel_bug_more_text_tl  
\c_msg_no_info_text_tl  
\c_msg_return_text_tl
```

Various pieces of text for use in messages, which are not changed by the code here although they could be to alter the language. Although these are marked as constants, they could be changed for printing errors in a different language.

```
\c_msg_on_line_tl
```

The ‘on line’ phrase for line numbers. Although marked as a constant, they could be changed for printing errors in a different language.

```
\c_msg_text_prefix_tl  
\c_msg_more_text_prefix_tl
```

Header information for storing the ‘paths’ to parts of a message. Although these are marked as constants, they could be changed for printing errors in a different language.

```
\l_msg_class_tl  
\l_msg_current_class_tl  
\l_msg_current_module_tl
```

Information about message method, used for filtering.

```
\l_msg_names_clist
```

List of all of the message names defined.

```
\l_msg_redirect_classes_prop  
\l_msg_redirect_names_prop
```

Re-direction lists containing the class of message to convert an different one.

```
\l_msg_redirect_classes_clist
```

List so that filtering does not loop.

Part XVIII

The l3xref package

Cross references

`\xref_set_label:n` `\xref_set_label:n {<name>}`

Sets a label in the text. Note that this function does not do anything else than setting the correct labels. In particular, it does not try to fix any spacing around the write node; this is a task for the `galley2` module.

`\xref_new:nn` `\xref_new:nn {<type>} {<value>}`

Defines a new cross reference type `<type>`. This defines the token list variable `\l_xref_curr_<type>_tl` with default value `<value>` which gets written fully expanded when `\xref_set_label:n` is called.

`\xref_deferred_new:nn` `\xref_deferred_new:nn {<type>} {<value>}`

Same as `\xref_new:n` except for this one, the value written happens when TeX ships out the page. Page numbers use this one obviously.

`\xref_get_value:nn *` `\xref_get_value:nn {<type>} {<name>}`

Extracts the cross reference information of type `<type>` for the label `<name>`. This operation is expandable.

Part XIX

The l3keyval package

Key-value parsing

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.

This module provides the low-level machinery for processing arbitrary key–value lists. The `l3keys` module provides a higher-level interface for managing run-time settings using key–value input, while other parts of L^AT_EX3 also use key–value input based on `l3keyval` (for example the `xtemplate` module).

82 Features of `l3keyval`

As `l3keyval` is a low-level module, its functions are restricted to converting a *keyval list* into keys and values for further processing. Each key and value (or key alone) has to be processed further by a function provided when `l3keyval` is called. Typically, this will be *via* one of the `\KV_process...` functions:

```
\KV_process_space_removal_sanitiz:NNn
\my_processor_function_one:n
\my_processor_function_two:nn
{ <keyval list> }
```

The two processor functions here handle the cases where there is only a key, and where there is both a key and value, respectively.

`l3keyval` parses key–value lists in a manner that does not double `#` tokens or expand any input. The module has processor functions which will sanitize the category codes of `=` and `,` tokens (for use in the document body) as well as faster versions which do not do this (for use inside code blocks). Spaces can be removed from each end of the key and value (again for the document body), again with faster code to be used where this is not necessary. Values which are wrapped in braces will have exactly one set removed, meaning that

```
key = {value here},
```

and

```
key = value here,
```

are treated as identical (assuming that space removal is in force). `l3keyval`

83 Functions for keyval processing

The `l3keyval` module should be accessed *via* a small set of external functions. These correctly set up the module internals for use by other parts of L^AT_EX3.

In all cases, two functions have to be supplied by the programmer to apply to the items from the <keyval list> after l3keyval has separated out the entries. The first function should take one argument, and will receive the names of keys for which no value was supplied. The second function should take two arguments: a key name and the associated value.

<code>\KV_process_space_removal_sanitize:NNn</code>	<code>\KV_process_space_removal_sanitize:NNn</code> <code>⟨function₁⟩ ⟨function₂⟩ {⟨keyval list⟩}</code>
---	---

Parses the <keyval list> splitting it into keys and associated values. Spaces are removed from the ends of both the key and value by this function, and the category codes of non-braced = and , tokens are normalised so that parsing is ‘category code safe’. After parsing is completed, <function₁> is used to process keys without values and <function₂> deals with keys which have associated values.

<code>\KV_process_space_removal_no_sanitize:NNn</code>	<code>\KV_process_space_removal_no_sanitize:NNn</code> <code>⟨function₁⟩ ⟨function₂⟩ {⟨keyval list⟩}</code>
--	--

Parses the <keyval list> splitting it into keys and associated values. Spaces are removed from the ends of both the key and value by this function, but category codes are not normalised. After parsing is completed, <function₁> is used to process keys without values and <function₂> deals with keys which have associated values.

<code>\KV_process_no_space_removal_no_sanitize:NNn</code>	<code>\KV_process_no_space_removal_no_sanitize:NNn</code> <code>⟨function₁⟩ ⟨function₂⟩ {⟨keyval list⟩}</code>
---	---

Parses the <keyval list> splitting it into keys and associated values. Spaces are *not* removed from the ends of the key and value, and category codes are *not* normalised. After parsing is completed, <function₁> is used to process keys without values and <function₂> deals with keys which have associated values.

<code>\l_KV_remove_one_level_of_braces_bool</code>	This boolean controls whether or not one level of braces is stripped from the key and value. The default value for this boolean is <code>true</code> so that exactly one level of braces is stripped. For certain applications it is desirable to keep the braces in which case the programmer just has to set the boolean false temporarily. Only applicable when spaces are being removed.
--	--

84 Internal functions

The remaining functions provided by l3keyval do not have any protection for nesting of one call to the module inside another. They should therefore not be called directly by other modules.

<code>\KV_parse_no_space_removal_no_sanitize:n</code>	<code>\KV_parse_no_space_removal_no_sanitize:n {⟨keyval list⟩}</code>
---	---

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are not removed in the parsing process and the category codes of `=` and `,` are not normalised.

<code>\KV_parse_space_removal_no_sanitize:n</code>	<code>\KV_parse_space_removal_no_sanitize:n {<keyval list>}</code>
--	--

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are removed in the parsing process from the ends of the key and value, but the category codes of `=` and `,` are not normalised.

<code>\KV_parse_space_removal_sanitize:n</code>	<code>\KV_parse_space_removal_sanitize:n {<keyval list>}</code>
---	---

Parses the keys and values, passing the results to `\KV_key_no_value_elt:n` and `\KV_key_value_elt:nn` as appropriate. Spaces are removed in the parsing process from the ends of the key and value and the category codes of `=` and `,` are normalised at the outer level (*i.e.* only unbraced tokens are affected).

<code>\KV_key_no_value_elt:n</code>	<code>\KV_key_no_value_elt:n {<key>}</code>
<code>\KV_key_value_elt:nn</code>	<code>\KV_key_value_elt:n {<key>} {<value>}</code>

Used by `\KV_parse...` functions to further process keys with no values and keys with values, respectively. The standard definitions are error functions: the programmer should provide appropriate definitions for both at point of use.

85 Variables and constants

<code>\c_KV_single_equal_sign_tl</code>	Constant token list to make finding <code>=</code> faster.
---	--

<code>\l_KV_tmpa_tl</code> <code>\l_KV_tmpp_tl</code>	Scratch token lists.
--	----------------------

<code>\l_KV_parse_tl</code> <code>\l_KV_currkey_tl</code> <code>\l_KV_currval_tl</code>	Token list variables for various parts of the parsed input.
---	---

Part XX

The l3keys package

Key–value support

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. For the user, the system normally results in input of the form

```
\PackageControlMacro{
  key-one = value one,
  key-two = value two
}
```

or

```
\PackageMacro[
  key-one = value one,
  key-two = value two
]{argument}.
```

For the programmer, the original keyval package gives only the most basic interface for this work. All key macros have to be created one at a time, and as a result the `kvoptions` and `xkeyval` packages have been written to extend the ease of creating keys. A very different approach has been provided by the `pgfkeys` package, which uses a key–value list to generate keys.

The `l3keys` package is aimed at creating a programming interface for key–value controls in L^AT_EX3. Keys are created using a key–value interface, in a similar manner to `pgfkeys`. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { module }
  key-one .code:n = code including parameter #1,
  key-two .tl_set:N = \l_module_store_tl
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { module }
  key-one = value one,
  key-two = value two
}
```

At a document level, `\keys_set:nn` is used within a document function. For L^AT_EX2_ε, a generic set up function could be created with

```
\newcommand*\SomePackageSetup[1]{%
  \@nameuse{keys_set:nn}{module}{#1}%
}
```

or to use key–value input as the optional argument for a macro:

```
\newcommand*\SomePackageMacro[2][]{%
  \begingroup
    \@nameuse{keys_set:nn}{module}{#1}%
    % Main code for \SomePackageMacro
  \endgroup
}
```

The same concepts using `xparse` for L^AT_EX3 use:

```
\DeclareDocumentCommand \SomePackageSetup { m } {
  \keys_set:nn { module } { #1 }
}
\DeclareDocumentCommand \SomePackageMacro { o m } {
  \group_begin:
    \keys_set:nn { module } { #1 }
    % Main code for \SomePackageMacro
  \group_end:
}
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As will be discussed in section 87, it is suggested that the character ‘/’ is reserved for sub-division of keys into logical groups. Macros are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_module_tmp_tl { key }
\keys_define:nn { module } {
  \l_module_tmp_tl .code:n = code
}
```

will create a key called `\l_module_tmp_tl`, and not one called `key`.

86 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 `\keys_define:nn` function does not skip spaces in the input, and does not check the category codes for `,` and `=` tokens. This means that it is intended for use with code blocks and other environments where spaces are ignored.

```

\bool_set:N
\bool_gset:N <key> .bool_set:N = <bool>

```

Defines $\langle key \rangle$ to set $\langle bool \rangle$ to $\langle value \rangle$ (which must be either `true` or `false`). Here, $\langle bool \rangle$ is a L^AT_EX3 boolean variable (*i.e.* created using `\bool_new:N`). If the variable does not exist, it will be created at the point that the key is set up.

```

\choice: <key> .choice:

```

Sets $\langle key \rangle$ to act as a multiple choice key. Each valid choice for $\langle key \rangle$ must then be created, as discussed in section [87.1](#).

```

\choice_code:n
\choice_code:x <key> .choice_code:n = <code>

```

Stores $\langle code \rangle$ for use when `.generate_choices:n` creates one or more choice sub-keys of the current key. Inside $\langle code \rangle$, `\l_keys_choice_tl` contains the name of the choice made, and `\l_keys_choice_int` is the position of the choice in the list given to `.generate_choices:n`. Choices are discussed in detail in section [87.1](#).

```

\code:n
\code:x <key> .code:n = <code>

```

Stores the $\langle code \rangle$ for execution when $\langle key \rangle$ is called. The $\langle code \rangle$ can include one parameter (`#1`), which will be the $\langle value \rangle$ given for the $\langle key \rangle$. The `.code:x` variant will expand $\langle code \rangle$ at the point where the $\langle key \rangle$ is created.

```

\default:n
\default:V <key> .default:n = <default>

```

Creates a $\langle default \rangle$ value for $\langle key \rangle$, which is used if no value is given. This will be used if only the key name is given, but not if a blank $\langle value \rangle$ is given:

```

\keys_define:nn { module } {
  key .code:n = Hello #1,
  key .default:n = World
}

```



```

}
\keys_set:nn { module} {
  key = Fred, % Prints 'Hello Fred'
  key,      % Prints 'Hello World'
  key = ,    % Prints 'Hello '
}

```

T_EXhackers note: The $\langle default \rangle$ is stored as a token list variable, and therefore should not contain unescaped # tokens.

<code>.dim_set:N</code>
<code>.dim_set:c</code>
<code>.dim_gset:N</code>
<code>.dim_gset:c</code>

 $\langle key \rangle .dim_set:N = \langle dimension \rangle$

Sets $\langle key \rangle$ to store the value it is given in $\langle dimension \rangle$. Here, $\langle dimension \rangle$ is a L^AT_EX3 dim variable (*i.e.* created using `\dim_new:N`) or a L^AT_EX2_ε `dimen` (*i.e.* created using `\newdimen`). If the variable does not exist, it will be created at the point that the key is set up.

<code>.fp_set:N</code>
<code>.fp_set:c</code>
<code>.fp_gset:N</code>
<code>.fp_gset:c</code>

 $\langle key \rangle .fp_set:N = \langle floating\ point \rangle$

Sets $\langle key \rangle$ to store the value it is given in $\langle floating\ point \rangle$. Here, $\langle floating\ point \rangle$ is a L^AT_EX3 fp variable (*i.e.* created using `\fp_new:N`). If the variable does not exist, it will be created at the point that the key is set up.

<code>.generate_choices:n</code>

 $\langle key \rangle .generate_choices:n = \langle comma\ list \rangle$

Makes $\langle key \rangle$ a multiple choice key, accepting the choices specified in $\langle comma\ list \rangle$. Each choice will execute code which should previously have been defined using `.choice_code:n` or `.choice_code:x`. Choices are discussed in detail in section 87.1.

<code>.int_set:N</code>
<code>.int_set:c</code>
<code>.int_gset:N</code>
<code>.int_gset:c</code>

 $\langle key \rangle .int_set:N = \langle integer \rangle$

Sets $\langle key \rangle$ to store the value it is given in $\langle integer \rangle$. Here, $\langle integer \rangle$ is a L^AT_EX3 int variable (*i.e.* created using `\int_new:N`) or a L^AT_EX2_ε `count` (*i.e.* created using `\newcount`).

If the variable does not exist, it will be created at the point that the key is set up.

<pre>.meta:n .meta:x</pre>	<pre>$\langle key \rangle$.meta:n = $\langle multiple\ keys \rangle$</pre>
----------------------------	---

Makes $\langle key \rangle$ a meta-key, which will set $\langle multiple\ keys \rangle$ in one go. If $\langle key \rangle$ is given with a value at the time the key is used, then the value will be passed through to the subsidiary $\langle keys \rangle$ for processing (as #1).

<pre>.skip_set:N .skip_set:c .skip_gset:N .skip_gset:c</pre>	<pre>$\langle key \rangle$.skip_set:N = $\langle skip \rangle$</pre>
--	---

Sets $\langle key \rangle$ to store the value it is given in $\langle skip \rangle$, which is created if it does not already exist. Here, $\langle skip \rangle$ is a L^AT_EX3 skip variable (*i.e.* created using `\skip_new:N`) or a L^AT_EX 2_ε skip (*i.e.* created using `\newskip`). If the variable does not exist, it will be created at the point that the key is set up.

<pre>.tl_set:N .tl_set:c .tl_set_x:N .tl_set_x:c .tl_gset:N .tl_gset:c .tl_gset_x:N .tl_gset_x:c</pre>	<pre>$\langle key \rangle$.tl_set:N = $\langle token\ list\ variable \rangle$</pre>
--	--

Sets $\langle key \rangle$ to store the value it is given in $\langle token\ list\ variable \rangle$, which is created if it does not already exist. Here, $\langle token\ list\ variable \rangle$ is a L^AT_EX3 tl variable (*i.e.* created using `\tl_new:N`) or a L^AT_EX 2_ε macro with no arguments (*i.e.* created using `\newcommand` or `\def`). If the variable does not exist, it will be created at the point that the key is set up. The x variants perform an x expansion at the time the $\langle value \rangle$ passed to the $\langle key \rangle$ is saved to the $\langle token\ list\ variable \rangle$.

<pre>.value_forbidden: .value_required:</pre>	<pre>$\langle key \rangle$.value_forbidden:</pre>
---	---

Flags for forbidding and requiring a $\langle value \rangle$ for $\langle key \rangle$. Giving a $\langle value \rangle$ for a $\langle key \rangle$ which has the `.value_forbidden:` property set will result in an error. In the same way, if a $\langle key \rangle$ has the `.value_required:` property set then a $\langle value \rangle$ must be given when the $\langle key \rangle$ is used.

87 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 { module } {  
  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.

87.1 Multiple choices

Multiple choices are created by setting the `.choice:` property:

```
\keys_define:nn { module } {  
  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 choices. Here, the keys can share the same code, and can be rapidly created using the `.choice_code:n` and `.generate_choices:n` properties:

```
\keys_define:nn { module } {  
  key .choice_code:n = {  
    You~gave~choice~‘‘\int_use:N \l_keys_choice_tl’’,~  
    which~is~in~position~  
    \int_use:N\l_keys_choice_int\space  
    in~the~list.  
  },  
  key .generate_choices:n = {  
    choice-a, choice-b, choice-c  
  }  
}
```

Following common computing practice, `\l_keys_choice_int` is indexed from 0 (as an offset), so that the value of `\l_keys_choice_int` for the first choice in a list will be zero. This means that `\l_keys_choice_int` can be used directly with `\if_case:w` and so on.

<code>\l_keys_choice_int</code> <code>\l_keys_choice_tl</code>

Inside the code block for a choice generated using `.generate_choice:`, 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 0.

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 { module } {
  key .choice:n,
  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 `.generate_choices:n` (*i.e.* anything might happen!).

88 Setting keys

<code>\keys_set:nn</code> <code>\keys_set:nV</code> <code>\keys_set:nv</code>

`\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. In contrast to `\keys_define:nn`, this function does check category codes and ignore spaces, and is therefore suitable for user input.

If a key is not known, `\keys_set:nn` will look for a special `unknown` key for the same module. This mechanism can be used to create new keys from user input.

```
\keys_define:nn { module } {
  unknown .code:n =
    You~tried~to~set~key~'\l_keys_path_tl'~to~'#1'
}
```

`\l_keys_key_tl` When processing an unknown key, the name of the key is available as `\l_keys_key_tl`. Note that this will have been processed using `\tl_to_str:N`. The value passed to the key (if any) is available as the macro parameter #1.

88.1 Examining keys: internal representation

`\keys_if_exist:nnTF` `\keys_if_exist:nnTF {<module>} {<key>} {<true code>} {<false code>}`

Tests if `<key>` exists for `<module>`, *i.e.* if any code has been defined for `<key>`.

TeXhackers note: The function works by testing for the existence of the internal function `\keys > <module>/<key>.cmd:n`.

`\keys_show:nn` `\keys_show:nn {<module>} {<key>}`

Shows the internal representation of a `<key>`.

TeXhackers note: Keys are stored as functions with names of the format `\keys > <module>/<key>.cmd:n`.

89 Internal functions

`\keys_bool_set:Nn` `\keys_bool_set:Nn <bool> {<scope>}`

Creates code to set `<bool>` when `<key>` is given, with setting using `<scope>` (empty or `g` for local or global, respectively). `<bool>` should be a L^AT_EX3 boolean variable.

`\keys_choice_code_store:x` `\keys_choice_code_store:x <code>`

Stores `<code>` for later use by `.generate_code:n`.

`\keys_choice_make:` `\keys_choice_make:`

Makes `<key>` a choice key.

`\keys_choices_generate:n` `\keys_choices_generate:n {<comma list>}`

Makes *comma list* choices for *key*.

`\keys_choice_find:n` `\keys_choice_find:n {<choice>}`
 Searches for *choice* as a sub-key of *key*.

`\keys_cmd_set:nn`
`\keys_cmd_set:nx` `\keys_cmd_set:nn {<path>} {<code>}`
 Creates a function for *path* using *code*.

`\keys_default_set:n`
`\keys_default_set:V` `\keys_default_set:n {<default>}`
 Sets *default* for *key*.

`\keys_define_elt:n`
`\keys_define_elt:nn` `\keys_define_elt:nn {<key>} {<value>}`
 Processing functions for key–value pairs when defining keys.

`\keys_define_key:n` `\keys_define_key:n {<key>}`
 Defines *key*.

`\keys_execute:` `\keys_execute:`
 Executes *key* (where the name of the *key* will be stored internally).

`\keys_execute_unknown:` `\keys_execute_unknown:`
 Handles unknown *key* names.

`\keys_if_value_requirement:nTF *` `\keys_if_value_requirement:nTF {<requirement>}`
`{<true code>} {<false code>}`
 Check if *requirement* applies to *key*.

`\keys_meta_make:n`
`\keys_meta_make:x` `\keys_meta_make:n {<keys>}`
 Makes *key* a meta-key to set *keys*.

`\keys_property_find:n` `\keys_property_find:n {<key>}`

Separates $\langle key \rangle$ from $\langle property \rangle$.

<code>\keys_property_new:nn</code> <code>\keys_property_new_arg:nn</code>	<code>\keys_property_new:nn {$\langle property \rangle$} {$\langle code \rangle$}</code>
--	--

Makes a new $\langle property \rangle$ expanding to $\langle code \rangle$. The `arg` version makes properties with one argument.

<code>\keys_property_undefine:n</code>	<code>\keys_property_undefine:n {$\langle property \rangle$}</code>
--	--

Deletes $\langle property \rangle$ of $\langle key \rangle$.

<code>\keys_set_elt:n</code> <code>\keys_set_elt:nn</code>	<code>\keys_set_elt:nn {$\langle key \rangle$} {$\langle value \rangle$}</code>
---	---

Processing functions for key–value pairs when setting keys.

<code>\keys_tmp:w</code>	<code>\keys_tmp:w $\langle args \rangle$</code>
--------------------------	--

Used to store $\langle code \rangle$ to execute a $\langle key \rangle$.

<code>\keys_value_or_default:n</code>	<code>\keys_value_or_default:n {$\langle value \rangle$}</code>
---------------------------------------	--

Sets `\l_keys_value_tl` to $\langle value \rangle$, or $\langle default \rangle$ if $\langle value \rangle$ was not given and if $\langle default \rangle$ is available.

<code>\keys_value_requirement:n</code>	<code>\keys_value_requirement:n {$\langle requirement \rangle$}</code>
--	---

Sets $\langle key \rangle$ to have $\langle requirement \rangle$ concerning $\langle value \rangle$.

<code>\keys_variable_set:NnNN</code> <code>\keys_variable_set:cnNN</code>	<code>\keys_variable_set:NnNN $\langle var \rangle$ $\langle type \rangle$ $\langle scope \rangle$ $\langle expansion \rangle$</code>
--	---

Sets $\langle key \rangle$ to assign $\langle value \rangle$ to $\langle variable \rangle$. The $\langle scope \rangle$ (blank for local, `g` for global) and $\langle type \rangle$ (`tl`, `int`, etc.) are given explicitly.

90 Variables and constants

<code>\c_keys_properties_root_tl</code> <code>\c_keys_root_tl</code>	The root paths for keys and properties, used to gen-
---	--

erate the names of the functions which store these items.

<code>\c_keys_value_forbidden_tl</code>
<code>\c_keys_value_required_tl</code>

Marker text containers: by storing the values the code can make comparisons slightly faster.

<code>\l_keys_choice_code_tl</code>

Used to transfer code from storage when making multiple choices.

<code>\l_keys_module_tl</code>
<code>\l_keys_path_tl</code>
<code>\l_keys_property_tl</code>

Various key paths need to be stored. These are flexible items that are set during the key reading process.

<code>\l_keys_no_value_bool</code>

A marker for ‘no value’ as key input.

<code>\l_keys_value_tl</code>

Holds the currently supplied value, in a token register as there may be # tokens.

Part XXI

In contrast to the `l3io` module, which deals with the lowest level of file management, the `l3file` module provides a higher level interface for handling file contents. This involves providing convenient wrappers around many of the functions in `l3io` to make them more generally accessible.

It is important to remember that \TeX will attempt to locate files using both the operating system path and entries in the \TeX file database (most \TeX systems use such a database). Thus the “current path” for \TeX is somewhat broader than that for other programs.

<code>\g_file_current_name_tl</code>

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 `\c_job_name_tl` at the start of a \LaTeX run and will be modified each time a file is read using `\file_input:n`.

<code>\file_if_exist:nTF</code>

`\file_if_exist:nTF {<file name>} {<true code>} {<false code>}`
 Searches for `<file name>` using the current \TeX search path and the additional paths controlled by `\file_path_include:n`. The branching versions then leave either `<true code>` or `<false code>` in the input stream, as appropriate to the truth of the test and the variant of the function chosen.

T_EXhackers note: The $\langle file\ name \rangle$ may contain both literal items and expandable content, which should on full expansion be the desired file name. The expansion occurs when T_EX searches for the file.

`\file_add_path:nN` $\file_add_path:nN\{\langle file\ name \rangle\}\ \langle tl\ var \rangle$

Searches for $\langle file\ name \rangle$ in the path as detailed for `\file_if_exist:nTF`, and if found sets the $\langle tl\ var \rangle$ the fully-qualified name of the file, *i.e.* the path and file name. If the file is not found then the $\langle tl\ var \rangle$ will be empty.

T_EXhackers note: The $\langle file\ name \rangle$ may contain both literal items and expandable content, which should on full expansion be the desired file name. The expansion occurs when T_EX searches for the file.

`\file_input:n` $\file_input:n\{\langle file\ name \rangle\}$

Searches for $\langle file\ name \rangle$ in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional L^AT_EX source. All files read are recorded for information and the file name stack is updated by this function.

T_EXhackers note: The $\langle file\ name \rangle$ may contain both literal items and expandable content, which should on full expansion be the desired file name. The expansion occurs when T_EX searches for the file.

`\file_path_include:n` $\file_path_include:n\{\langle path \rangle\}$

Adds $\langle path \rangle$ to the list of those used to search for files by the `\file_input:n` and `\file_if_exist:n` function. The assignment is local.

`\file_path_remove:n` $\file_path_remove:n\{\langle path \rangle\}$

Removes $\langle path \rangle$ from the list of those used to search for files by the `\file_input:n` and `\file_if_exist:n` function. The assignment is local.

`\file_list:` $\file_list:$

This function will list all files loaded using `\file_input:n` in the log file.

Part XXII

The l3fp package

Floating point arithmetic

91 Floating point numbers

A floating point number is one which is stored as a mantissa and a separate exponent. This module implements arithmetic using radix 10 floating point numbers. This means that the mantissa should be a real number in the range $1 \leq |x| < 10$, with the exponent given as an integer between -99 and 99 . In the input, the exponent part is represented starting with an `e`. As this is a low-level module, error-checking is minimal. Numbers which are too large for the floating point unit to handle will result in errors, either from `TeX` or from `LATEX`. The `LATEX` code does not check that the input will not overflow, hence the possibility of a `TeX` error. On the other hand, numbers which are too small will be dropped, which will mean that extra decimal digits will simply be lost.

When parsing numbers, any missing parts will be interpreted as zero. So for example

```
\fp_set:Nn \l_my_fp { }  
\fp_set:Nn \l_my_fp { . }  
\fp_set:Nn \l_my_fp { - }
```

will all be interpreted as zero values without raising an error.

Operations which give an undefined result (such as division by 0) will not lead to errors. Instead special marker values are returned, which can be tested for using for example `\fp_if_undefined:N(TF)`. In this way it is possible to work with asymptotic functions without first checking the input. If these special values are carried forward in calculations they will be treated as 0.

Floating point numbers are stored in the `fp` floating point variable type. This has a standard range of functions for variable management.

91.1 Constants

`\c_e_fp` The value of the base of natural numbers, e .

`\c_one_fp` A floating point variable with permanent value 1: used for speeding up some comparisons.

`\c_pi_fp` The value of π .

`\c_undefined_fp` A special marker floating point variable representing the result of an operation which does not give a defined result (such as division by 0).

`\c_zero_fp` A permanently zero floating point variable.

91.2 Floating-point variables

<code>\fp_new:N</code> <code>\fp_new:c</code>
--

`\fp_new:N` *<floating point variable>*

Creates a new *<floating point variable>* or raises an error if the name is already taken. The declaration global. The *<floating point>* will initially be set to `+0.000000000e0` (the zero floating point).

<code>\fp_const:Nn</code> <code>\fp_const:cn</code>
--

`\fp_const:Nn` *<floating point variable>* {*<value>*}

Creates a new constant *<floating point variable>* or raises an error if the name is already taken. The value of the *<floating point variable>* will be set globally to the *<value>*.

<code>\fp_set_eq:NN</code> <code>\fp_set_eq:cN</code> <code>\fp_set_eq:Nc</code> <code>\fp_set_eq:cc</code>
--

`\fp_set_eq:NN` *<fp var1>* *<fp var2>*

Sets the value of *<floating point variable1>* equal to that of *<floating point variable2>*. This assignment is restricted to the current \TeX group level.

<code>\fp_gset_eq:NN</code> <code>\fp_gset_eq:cN</code> <code>\fp_gset_eq:Nc</code> <code>\fp_gset_eq:cc</code>
--

`\fp_gset_eq:NN` *<fp var1>* *<fp var2>*

Sets the value of *<floating point variable1>* equal to that of *<floating point variable2>*. This assignment is global and so is not limited by the current \TeX group level.

<code>\fp_zero:N</code> <code>\fp_zero:c</code>
--

`\fp_zero:N` *<floating point variable>*

Sets the *<floating point variable>* to `+0.000000000e0` within the current scope.

<code>\fp_gzero:N</code> <code>\fp_gzero:c</code>
--

`\fp_gzero:N` *<floating point variable>*

Sets the *<floating point variable>* to `+0.000000000e0` globally.

<code>\fp_set:Nn</code> <code>\fp_set:cn</code>
--

`\fp_set:Nn` *<floating point variable>* {*<value>*}

Sets the *<floating point variable>* variable to *<value>* within the scope of the current \TeX

group.

<code>\fp_gset:Nn</code>

<code>\fp_gset:cn</code>

`\fp_gset:Nn` *<floating point variable>* {*<value>*}

Sets the *<floating point variable>* variable to *<value>* globally.

<code>\fp_set_from_dim:Nn</code>

<code>\fp_set_from_dim:cn</code>

`\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>*. The assignment is local.

<code>\fp_gset_from_dim:Nn</code>

<code>\fp_gset_from_dim:cn</code>

`\fp_gset_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>*. The assignment is global.

<code>\fp_use:N *</code>

<code>\fp_use:c *</code>

`\fp_use:N` *<floating point variable>*

Inserts the value of the *<floating point variable>* into the input stream. The value will be given as a real number without any exponent part, and will always include a decimal point. For example,

```
\fp_new:Nn \test
\fp_set:Nn \test { 1.234 e 5 }
\fp_use:N \test
```

will insert ‘12345.00000’ into the input stream. As illustrated, a floating point will always be inserted with ten significant digits given. Very large and very small values will include additional zeros for place value.

<code>\fp_show:N</code>

<code>\fp_show:c</code>

`\fp_show:N` *<floating point variable>*

Displays the content of the *<floating point variable>* on the terminal.

91.3 Conversion to other formats

It is useful to be able to convert floating point variables to other forms. These functions are expandable, so that the material can be used in a variety of contexts. The `\fp_use:N`

function should also be consulted in this context, as it will insert the value of the floating point variable as a real number.

<code>\fp_to_dim:N *</code>
<code>\fp_to_dim:c *</code>

`\fp_to_dim:N <floating point variable>`

Inserts the value of the *<floating point variable>* into the input stream converted into a dimension in points.

<code>\fp_to_int:N *</code>
<code>\fp_to_int:c *</code>

`\fp_to_int:N <floating point variable>`

Inserts the integer value of the *<floating point variable>* into the input stream. The decimal part of the number will not be included, but will be used to round the integer.

<code>\fp_to_tl:N *</code>
<code>\fp_to_tl:c *</code>

`\fp_to_tl:N <floating point variable>`

Inserts a representation of the *<floating point variable>* into the input stream as a token list. The representation follows the conventions of a pocket calculator:

Floating point value	Representation
1.234000000000e0	1.234
-1.234000000000e0	-1.234
1.234000000000e3	1234
1.234000000000e13	1234e13
1.234000000000e-1	0.1234
1.234000000000e-2	0.01234
1.234000000000e-3	1.234e-3

Notice that trailing zeros are removed in this process, and that numbers which do not require a decimal part do *not* include a decimal marker.

91.4 Rounding floating point values

The module can round floating point values to either decimal places or significant figures using the usual method in which exact halves are rounded up.

<code>\fp_round_figures:Nn</code>
<code>\fp_round_figures:cn</code>

`\fp_round_figures:Nn <floating point variable> {<target>}`

Rounds the $\langle floating\ point\ variable \rangle$ to the $\langle target \rangle$ number of significant figures (an integer expression). The rounding is carried out locally.

$\backslash\text{fp_ground_figures:Nn}$
$\backslash\text{fp_ground_figures:cn}$

 $\backslash\text{fp_ground_figures:Nn}$ $\langle floating\ point\ variable \rangle$ $\{\langle target \rangle\}$

Rounds the $\langle floating\ point\ variable \rangle$ to the $\langle target \rangle$ number of significant figures (an integer expression). The rounding is carried out globally.

$\backslash\text{fp_round_places:Nn}$
$\backslash\text{fp_round_places:cn}$

 $\backslash\text{fp_round_places:Nn}$ $\langle floating\ point\ variable \rangle$ $\{\langle target \rangle\}$

Rounds the $\langle floating\ point\ variable \rangle$ to the $\langle target \rangle$ number of decimal places (an integer expression). The rounding is carried out locally.

$\backslash\text{fp_ground_places:Nn}$
$\backslash\text{fp_ground_places:cn}$

 $\backslash\text{fp_ground_places:Nn}$ $\langle floating\ point\ variable \rangle$ $\{\langle target \rangle\}$

Rounds the $\langle floating\ point\ variable \rangle$ to the $\langle target \rangle$ number of decimal places (an integer expression). The rounding is carried out globally.

91.5 Tests on floating-point values

$\backslash\text{fp_if_undefined_p:N}$ \star
$\backslash\text{fp_if_undefined:N}\underline{\text{TF}}$ \star

 $\backslash\text{fp_if_undefined_p:N}$ $\langle fixed\text{-}point \rangle$
 $\backslash\text{fp_if_undefined:N}\underline{\text{TF}}$ $\langle fixed\text{-}point \rangle$
 $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests if $\langle floating\ point \rangle$ is undefined (*i.e.* equal to the special $\backslash\text{c_undefined_fp}$ variable). The branching versions then leave either $\langle true\ code \rangle$ or $\langle false\ code \rangle$ in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

$\backslash\text{fp_if_zero_p:N}$ \star
$\backslash\text{fp_if_zero:N}\underline{\text{TF}}$ \star

 $\backslash\text{fp_if_zero_p:N}$ $\langle fixed\text{-}point \rangle$
 $\backslash\text{fp_if_zero:N}\underline{\text{TF}}$ $\langle fixed\text{-}point \rangle$ $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

Tests if $\langle floating\ point \rangle$ is equal to zero (*i.e.* equal to the special $\backslash\text{c_zero_fp}$ variable). The branching versions then leave either $\langle true\ code \rangle$ or $\langle false\ code \rangle$ in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The logical truth of the test is left in the input stream by the predicate version.

$\backslash\text{fp_compare:nNn}\underline{\text{TF}}$

 $\backslash\text{fp_compare:nNn}\underline{\text{TF}}$ $\{\langle floating\ point_1 \rangle\}$ $\langle relation \rangle$ $\{\langle floating\ point_2 \rangle\}$
 $\{\langle true\ code \rangle\}$ $\{\langle false\ code \rangle\}$

This function compared the two $\langle floating\ point \rangle$ values, which may be stored as fp variables, using the $\langle relation \rangle$:

Equal	=
Greater than	>
Less than	<

Either *⟨true code⟩* or *⟨false code⟩* is then left in the input stream, as appropriate to the truth of the test and the variant of the function chosen. The tests treat undefined floating points as zero as the comparison is intended for real numbers only.

91.6 Unary operations

The unary operations alter the value stored within an **fp** variable.

<code>\fp_abs:N</code>
<code>\fp_abs:c</code>

`\fp_abs:N` *⟨floating point variable⟩*

Converts the *⟨floating point variable⟩* to its absolute value, assigning the result within the current T_EX group.

<code>\fp_gabs:N</code>
<code>\fp_gabs:c</code>

`\fp_gabs:N` *⟨floating point variable⟩*

Converts the *⟨floating point variable⟩* to its absolute value, assigning the result globally.

<code>\fp_neg:N</code>
<code>\fp_neg:c</code>

`\fp_neg:N` *⟨floating point variable⟩*

Reverse the sign of the *⟨floating point variable⟩*, assigning the result within the current T_EX group.

<code>\fp_gneg:N</code>
<code>\fp_gneg:c</code>

`\fp_gneg:N` *⟨floating point variable⟩*

Reverse the sign of the *⟨floating point variable⟩*, assigning the result globally.

91.7 Arithmetic operations

Binary arithmetic operations act on the value stored in an **fp**, so for example

```
\fp_set:Nn \l_my_fp { 1.234 }
\fp_sub:Nn \l_my_fp { 5.678 }
```

sets `\l_my_fp` to the result of $1.234 - 5.678$ (i.e. -4.444).

<code>\fp_add:Nn</code>	<code>\fp_add:Nn <floating point> {<value>}</code>
<code>\fp_add:cn</code>	

Adds the *<value>* to the *<floating point>*, making the assignment within the current `TeX` group level.

<code>\fp_gadd:Nn</code>	<code>\fp_gadd:Nn <floating point> {<value>}</code>
<code>\fp_gadd:cn</code>	

Adds the *<value>* to the *<floating point>*, making the assignment globally.

<code>\fp_sub:Nn</code>	<code>\fp_sub:Nn <floating point> {<value>}</code>
<code>\fp_sub:cn</code>	

Subtracts the *<value>* from the *<floating point>*, making the assignment within the current `TeX` group level.

<code>\fp_gsub:Nn</code>	<code>\fp_gsub:Nn <floating point> {<value>}</code>
<code>\fp_gsub:cn</code>	

Subtracts the *<value>* from the *<floating point>*, making the assignment globally.

<code>\fp_mul:Nn</code>	<code>\fp_mul:Nn <floating point> {<value>}</code>
<code>\fp_mul:cn</code>	

Multiples the *<floating point>* by the *<value>*, making the assignment within the current `TeX` group level.

<code>\fp_gmul:Nn</code>	<code>\fp_gmul:Nn <floating point> {<value>}</code>
<code>\fp_gmul:cn</code>	

Multiples the *<floating point>* by the *<value>*, making the assignment globally.

<code>\fp_div:Nn</code>	<code>\fp_div:Nn <floating point> {<value>}</code>
<code>\fp_div:cn</code>	

Divides the *<floating point>* by the *<value>*, making the assignment within the current `TeX` group level. If the *<value>* is zero, the *<floating point>* will be set to `\c_undefined_fp`. The assignment is local.

<code>\fp_gdiv:Nn</code>	<code>\fp_gdiv:Nn <floating point> {<value>}</code>
<code>\fp_gdiv:cn</code>	

Divides the *<floating point>* by the *<value>*, making the assignment globally. If the *<value>* is zero, the *<floating point>* will be set to `\c_undefined_fp`. The assignment is global.

91.8 Power operations

<code>\fp_pow:Nn</code> <code>\fp_pow:cn</code>
--

`\fp_pow:Nn` *<floating point>* {*<value>*}

Raises the *<floating point>* to the given *<value>*. If the *<floating point>* is negative, then the *<value>* should be either a positive real number or a negative integer. If the *<floating point>* is positive, then the *<value>* may be any real value. Mathematically invalid operations such as 0^0 will give set the *<floating point>* to to `\c_undefined_fp`. The assignment is local.

<code>\fp_gpow:Nn</code> <code>\fp_gpow:cn</code>
--

`\fp_gpow:Nn` *<floating point>* {*<value>*}

Raises the *<floating point>* to the given *<value>*. If the *<floating point>* is negative, then the *<value>* should be either a positive real number or a negative integer. If the *<floating point>* is positive, then the *<value>* may be any real value. Mathematically invalid operations such as 0^0 will give set the *<floating point>* to to `\c_undefined_fp`. The assignment is global.

91.9 Exponential and logarithm functions

<code>\fp_exp:Nn</code> <code>\fp_exp:cn</code>
--

`\fp_exp:Nn` *<floating point>* {*<value>*}

Calculates the exponential of the *<value>* and assigns this to the *<floating point>*. The assignment is local.

<code>\fp_gexp:Nn</code> <code>\fp_gexp:cn</code>
--

`\fp_gexp:Nn` *<floating point>* {*<value>*}

Calculates the exponential of the *<value>* and assigns this to the *<floating point>*. The assignment is global.

<code>\fp_ln:Nn</code> <code>\fp_ln:cn</code>
--

`\fp_ln:Nn` *<floating point>* {*<value>*}

Calculates the natural logarithm of the *<value>* and assigns this to the *<floating point>*. The assignment is local.

<code>\fp_gln:Nn</code> <code>\fp_gln:cn</code>
--

`\fp_gln:Nn` *<floating point>* {*<value>*}

Calculates the natural logarithm of the *<value>* and assigns this to the *<floating point>*. The assignment is global.

91.10 Trigonometric functions

The trigonometric functions all work in radians. They accept a maximum input value of 100 000 000, as there are issues with range reduction and very large input values.

<code>\fp_sin:Nn</code>
<code>\fp_sin:cn</code>

`\fp_sin:Nn` *<floating point>* {*<value>*}

Assigns the sine of the *<value>* to the *<floating point>*. The *<value>* should be given in radians. The assignment is local.

<code>\fp_gsin:Nn</code>
<code>\fp_gsin:cn</code>

`\fp_gsin:Nn` *<floating point>* {*<value>*}

Assigns the sine of the *<value>* to the *<floating point>*. The *<value>* should be given in radians. The assignment is global.

<code>\fp_cos:Nn</code>
<code>\fp_cos:cn</code>

`\fp_cos:Nn` *<floating point>* {*<value>*}

Assigns the cosine of the *<value>* to the *<floating point>*. The *<value>* should be given in radians. The assignment is local.

<code>\fp_gcos:Nn</code>
<code>\fp_gcos:cn</code>

`\fp_gcos:Nn` *<floating point>* {*<value>*}

Assigns the cosine of the *<value>* to the *<floating point>*. The *<value>* should be given in radians. The assignment is global.

<code>\fp_tan:Nn</code>
<code>\fp_tan:cn</code>

`\fp_tan:Nn` *<floating point>* {*<value>*}

Assigns the tangent of the *<value>* to the *<floating point>*. The *<value>* should be given in radians. The assignment is local.

<code>\fp_gtan:Nn</code>
<code>\fp_gtan:cn</code>

`\fp_gtan:Nn` *<floating point>* {*<value>*}

Assigns the tangent of the *<value>* to the *<floating point>*. The *<value>* should be given in radians. The assignment is global.

91.11 Notes on the floating point unit

As calculation of the elemental transcendental functions is computationally expensive compared to storage of results, after calculating a trigonometric function, exponent, *etc.* the module stored the result for reuse. Thus the performance of the module for

repeated operations, most probably trigonometric functions, should be much higher than if the values were re-calculated every time they were needed.

Anyone with experience of programming floating point calculations will know that this is a complex area. The aim of the unit is to be accurate enough for the likely applications in a typesetting context. The arithmetic operations are therefore intended to provide ten digit accuracy with the last digit accurate to ± 1 . The elemental transcendental functions may not provide such high accuracy in every case, although the design aim has been to provide 10 digit accuracy for cases likely to be relevant in typesetting situations. A good overview of the challenges in this area can be found in J.-M. Muller, *Elementary functions: algorithms and implementation*, 2nd edition, Birkhäuser Boston, New York, USA, 2006.

The internal representation of numbers is tuned to the needs of the underlying \TeX system. This means that the format is somewhat different from that used in, for example, computer floating point units. Programming in \TeX makes it most convenient to use a radix 10 system, using \TeX `count` registers for storage and taking advantage where possible of delimited arguments.

Part XXIII

The $\text{\texttt{l3luatex}}$ package Lua \TeX -specific functions

92 Breaking out to Lua

The Lua \TeX 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 \pdf\TeX or \Xe\TeX these will raise an error: use $\text{\engine_if_luatex:T}$ to avoid this. Details of coding the Lua \TeX engine are detailed in the Lua \TeX manual.

\lua_now:n *
\lua_now:x *

 \lua_now:n { $\langle token\ list \rangle$ }

The $\langle token\ list \rangle$ 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 $\langle Lua\ input \rangle$ is passed to the Lua interpreter for processing. Each \lua_now:n block is treated by Lua as a separate chunk. The Lua interpreter will execute the $\langle Lua\ input \rangle$ immediately, and in an expandable manner.

\TeX hackers note: \lua_now:x is the Lua \TeX primitive \directlua renamed.

<code>\lua_shipout:n</code>	<code>\lua_shipout:x {⟨token list⟩}</code>
<code>\lua_shipout:x</code>	

The $\langle token\ list \rangle$ 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 $\langle Lua\ input \rangle$ is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the $\langle Lua\ input \rangle$ during the page-building routine: no \TeX expansion of the $\langle Lua\ input \rangle$ will occur at this stage.

\TeX hackers note: At a \TeX level, the $\langle Lua\ input \rangle$ is stored as a ‘whatsit’.

<code>\lua_shipout_x:n</code>	<code>\lua_shipout:n {⟨token list⟩}</code>
<code>\lua_shipout_x:x</code>	

The $\langle token\ list \rangle$ 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 $\langle Lua\ input \rangle$ is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the $\langle Lua\ input \rangle$ during the page-building routine: the $\langle Lua\ input \rangle$ 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).

\TeX hackers note: `\lua_sjhipout_x:n` is the Lua \TeX primitive `\lattelua` named using the \LaTeX 3 scheme.

At a \TeX level, the $\langle Lua\ input \rangle$ is stored as a ‘whatsit’.

93 Category code tables

As well as providing methods to break out into Lua, there are places where additional \LaTeX 3 functions are provided by the Lua \TeX engine. In particular, Lua \TeX 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 `\ExplSyntaxOn` and `\ExplSyntaxOff` when using the Lua \TeX engine.

<code>\cctab_new:N</code>	<code>\cctab_new:N ⟨category code table⟩</code>
---------------------------	---

Creates a new category code table, initially with the codes as used by `\Init\TeX`.

<code>\cctab_gset:Nn</code>	<code>\cctab_gset:Nn ⟨category code table⟩ {⟨category code set up⟩}</code>
-----------------------------	--

Sets the $\langle category\ code\ table \rangle$ to apply the category codes which apply when the prevailing

regime is modified by the $\langle category\ code\ set\ up \rangle$. Thus within a standard code block the starting point will be the code applied by `\c_code_cctab`. The assignment of the table is global: the underlying primitive does not respect grouping.

`\cctab_begin:N` `\cctab_begin:N` $\langle category\ code\ table \rangle$

Switches the category codes in force to those stored in the $\langle category\ code\ table \rangle$. The prevailing codes before the function is called are added to a stack, for use with `\cctab_end:`.

`\cctab_end:` `\cctab_end:`

Ends the scope of a $\langle category\ code\ table \rangle$ started using `\cctab_begin:N`, retuning the codes to those in force before the matching `\cctab_begin:N` was used.

`\c_code_cctab` Category code table for the code environment. This does not include setting the behaviour of the line-end character, which is only altered by `\ExplSyntaxOn`.

`\c_document_cctab` Category code table for a standard L^AT_EX document. This does not include setting the behaviour of the line-end character, which is only altered by `\ExplSyntaxOff`.

`\c_initex_cctab` Category code table as set up by IniT_EX.

`\c_other_cctab` Category code table where all characters have category code 12 (other).

`\c_string_cctab` Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).

Part XXIV

Implementation

94 l3names implementation

This is the base part of L^AT_EX3 defining things like `catcodes` and redefining the T_EX primitives, as well as setting up the code to load `expl3` modules in L^AT_EX 2_ε.

94.1 Internal functions

<code>\ExplSyntaxStatus</code>
<code>\ExplSyntaxPopStack</code>
<code>\ExplSyntaxStack</code>

Functions used to track the state of the catcode regime.

<code>\@pushfilename</code>
<code>\@popfilename</code>

Re-definitions of L^AT_EX's file-loading functions to support `\ExplSyntax`.

94.2 Bootstrap code

The very first thing to do is to bootstrap the Init_ETeX system so that everything else will actually work. T_EX does not start with some pretty basic character codes set up.

```
1 <!*package>
2 \catcode '\{ = 1 \relax
3 \catcode '\} = 2 \relax
4 \catcode '\# = 6 \relax
5 \catcode '\^ = 7 \relax
6 </!package>
```

Tab characters should not show up in the code, but to be on the safe side.

```
7 <!*package>
8 \catcode '\^^I = 10 \relax
9 </!package>
```

For Lua_ETeX the extra primitives need to be enabled before they can be use. No `\ifdefined` yet, so do it the old-fashioned way. The primitive `\strcmp` is simulated using some Lua code, which currently has to be applied to every job as the Lua code is not part of the format. Thanks to Taco Hoekwater for this code. The odd `\csname` business is needed so that the later deletion code will work.

```
10 <!*package>
11 \begingroup\expandafter\expandafter\expandafter\endgroup
12 \expandafter\ifx\csname directlua\endcsname\relax
13 \else
14 \directlua
15 {
16 tex.enableprimitives('',tex.extraprimitives ())
17 lua.bytecode[1] = function ()
18 function strcmp (A, B)
19 if A == B then
20 tex.write("0")
21 elseif A < B then
```

```

22         tex.write("-1")
23     else
24         tex.write("1")
25     end
26 end
27 end
28 lua.bytecode[1]()
29 }
30 \everyjob\expandafter
31 {\csname tex_directlua:D\endcsname{lua.bytecode[1]()}}
32 \long\edef\pdfstrcmp#1#2%
33 {%
34     \expandafter\noexpand\csname tex_directlua:D\endcsname
35     {%
36         strcmp(%
37             "\noexpand\luaescapestring{#1}",%
38             "\noexpand\luaescapestring{#2}"%
39         )%
40     }%
41 }
42 \fi
43 \</!package>

```

When loaded as a package this can all be handed off to other L^AT_EX 2_ε code.

```

44 <*package>
45 \def\@tempa{%
46     \def\@tempa{%
47         \RequirePackage{luatex}%
48         \RequirePackage{pdftexcmds}%
49         \let\pdfstrcmp\pdf@strcmp
50     }
51 \begingroup\expandafter\expandafter\expandafter\endgroup
52 \expandafter\ifx\csname directlua\endcsname\relax
53 \else
54     \expandafter\@tempa
55 \fi
56 \</package>

```

X_εT_EX calls the primitive `\strcmp`, so there needs to be a check for that too.

```

57 \begingroup\expandafter\expandafter\expandafter\endgroup
58 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
59     \let\pdfstrcmp\strcmp
60 \fi

```

94.3 Requirements

Currently, the code requires the ε -T_EX primitives and functionality equivalent to `\pdfstrcmp`. Any package which provides the later will provide the former, so the test

can be done only for `\pdfstrcmp`.

```

61 \begingroup\expandafter\expandafter\expandafter\endgroup
62 \expandafter\ifx\csname pdfstrcmp\endcsname\relax
63 <*package>
64 \PackageError{l3names}{Required primitive not found: \protect\pdfstrcmp}
65 {%
66     LaTeX3 requires the e-TeX primitives and
67     \string\pdfstrcmp.\MessageBreak
68     These are available in engine versions: \MessageBreak
69     - pdfTeX 1.30 \MessageBreak
70     - XeTeX 0.9994 \MessageBreak
71     - LuaTeX 0.60 \MessageBreak
72     or later. \MessageBreak
73     \MessageBreak
74     Loading of l3names will abort!
75 }
76 </package>
77 <!*package>
78 \newlinechar'\^^J\relax
79 \errhelp{%
80     LaTeX3 requires the e-TeX primitives and
81     \string\pdfstrcmp. ^^J
82     These are available in engine versions: ^^J
83     - pdfTeX 1.30 ^^J
84     - XeTeX 0.9994 ^^J
85     - LuaTeX 0.60 ^^J
86     or later. ^^J
87     For pdfTeX and XeTeX the '-etex' command-line switch is also
88     needed.^^J
89     ^^J
90     Format building will abort!
91 }
92 </!package>
93 \expandafter\endinput
94 \fi

```

94.4 Catcode assignments

Catcodes for `\begingroup`, `\endgroup`, macro parameter, superscript, and `\tab`, are all assigned before the start of the documented code. (See the beginning of `l3names.dtx`.)

Reason for `\endlinechar=32` is that a line ending with a backslash will be interpreted as the token `_` which seems most natural and since spaces are ignored it works as we intend elsewhere.

Before we do this we must however record the settings for the catcode regime as it was when we start changing it.


```

95 <*initex | package>
96 \protected\edef\ExplSyntaxOff{
97   \unexpanded{\ifodd \ExplSyntaxStatus\relax
98     \def\ExplSyntaxStatus{0}
99   }
100   \catcode 126=\the \catcode 126 \relax
101   \catcode 32=\the \catcode 32 \relax
102   \catcode 9=\the \catcode 9 \relax
103   \endlinechar =\the \endlinechar \relax
104   \catcode 95=\the \catcode 95 \relax
105   \catcode 58=\the \catcode 58 \relax
106   \catcode 124=\the \catcode 124 \relax
107   \catcode 38=\the \catcode 38 \relax
108   \catcode 94=\the \catcode 94 \relax
109   \catcode 34=\the \catcode 34 \relax
110   \noexpand\fi
111 }
112 \catcode126=10\relax % tilde is a space char.
113 \catcode32=9\relax   % space is ignored
114 \catcode9=9\relax    % tab also ignored
115 \endlinechar=32\relax % endline is space
116 \catcode95=11\relax  % underscore letter
117 \catcode58=11\relax  % colon letter
118 \catcode124=12\relax % vert bar, other
119 \catcode38=4\relax   % ampersand, alignment token
120 \catcode34=12\relax  % doublequote, other
121 \catcode94=7\relax   % caret, math superscript

```

94.5 Setting up primitive names

Here is the function that renames \TeX 's primitives.

Normally the old name is left untouched, but the possibility of undefining the original names is made available by `docstrip` and package options. If nothing else, this gives a way of checking what ‘old code’ a package depends on...

If the package option ‘`removeoldnames`’ is used then some trick code is run after the end of this file, to skip past the code which has been inserted by $\text{\LaTeX 2}_{\epsilon}$ to manage the file name stack, this code would break if run once the \TeX primitives have been undefined. (What a surprise!) **The option has been temporarily disabled.**

To get things started, give a new name for `\let`.

```

122 \let \tex_let:D \let
123 </initex | package>

```

and now an internal function to possibly remove the old name: for the moment.

```

124 <*initex>
125 \long \def \name_undefine:N #1 {

```

```

126 \tex_let:D #1 \c_undefined
127 }
128 </initex>

129 <*package>
130 \DeclareOption{removeoldnames}{
131   \long\def\name_undefine:N#1{
132     \tex_let:D#1\c_undefined}}

133 \DeclareOption{keepoldnames}{
134   \long\def\name_undefine:N#1{}}

135 \ExecuteOptions{keepoldnames}

136 \ProcessOptions
137 </package>

```

The internal function to give the new name and possibly undefine the old name.

```

138 <*initex | package>
139 \long \def \name_primitive:NN #1#2 {
140   \tex_let:D #2 #1
141   \name_undefine:N #1
142 }

```

94.6 Reassignment of primitives

In the current incarnation of this package, all TeX primitives are given a new name of the form `\tex_oldname:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```

143 \name_primitive:NN \          \tex_space:D
144 \name_primitive:NN \/        \tex_italiccor:D
145 \name_primitive:NN \-        \tex_hyphen:D

```

Now all the other primitives.

```

146 \name_primitive:NN \let      \tex_let:D
147 \name_primitive:NN \def      \tex_def:D
148 \name_primitive:NN \edef     \tex_edef:D
149 \name_primitive:NN \gdef     \tex_gdef:D
150 \name_primitive:NN \xdef     \tex_xdef:D
151 \name_primitive:NN \chardef  \tex_chardef:D
152 \name_primitive:NN \countdef \tex_countdef:D
153 \name_primitive:NN \dimendef \tex_dimendef:D
154 \name_primitive:NN \skipdef  \tex_skipdef:D
155 \name_primitive:NN \muskipdef \tex_muskipdef:D
156 \name_primitive:NN \mathchardef \tex_mathchardef:D
157 \name_primitive:NN \toksdef  \tex_toksdef:D

```

158	\name_primitive:NN	\futurelet	\tex_futurelet:D
159	\name_primitive:NN	\advance	\tex_advance:D
160	\name_primitive:NN	\divide	\tex_divide:D
161	\name_primitive:NN	\multiply	\tex_multiply:D
162	\name_primitive:NN	\font	\tex_font:D
163	\name_primitive:NN	\fam	\tex_fam:D
164	\name_primitive:NN	\global	\tex_global:D
165	\name_primitive:NN	\long	\tex_long:D
166	\name_primitive:NN	\outer	\tex_outer:D
167	\name_primitive:NN	\setlanguage	\tex_setlanguage:D
168	\name_primitive:NN	\globaldefs	\tex_globaldefs:D
169	\name_primitive:NN	\afterassignment	\tex_afterassignment:D
170	\name_primitive:NN	\aftergroup	\tex_aftergroup:D
171	\name_primitive:NN	\expandafter	\tex_expandafter:D
172	\name_primitive:NN	\noexpand	\tex_noexpand:D
173	\name_primitive:NN	\begingroup	\tex_begingroup:D
174	\name_primitive:NN	\endgroup	\tex_endgroup:D
175	\name_primitive:NN	\halign	\tex_halign:D
176	\name_primitive:NN	\valign	\tex_valign:D
177	\name_primitive:NN	\cr	\tex_cr:D
178	\name_primitive:NN	\crrcr	\tex_crrcr:D
179	\name_primitive:NN	\noalign	\tex_noalign:D
180	\name_primitive:NN	\omit	\tex_omit:D
181	\name_primitive:NN	\span	\tex_span:D
182	\name_primitive:NN	\tabskip	\tex_tabskip:D
183	\name_primitive:NN	\everycr	\tex_everycr:D
184	\name_primitive:NN	\if	\tex_if:D
185	\name_primitive:NN	\ifcase	\tex_ifcase:D
186	\name_primitive:NN	\ifcat	\tex_ifcat:D
187	\name_primitive:NN	\ifnum	\tex_ifnum:D
188	\name_primitive:NN	\ifodd	\tex_ifodd:D
189	\name_primitive:NN	\ifdim	\tex_ifdim:D
190	\name_primitive:NN	\ifeof	\tex_ifeof:D
191	\name_primitive:NN	\ifhbox	\tex_ifhbox:D
192	\name_primitive:NN	\ifvbox	\tex_ifvbox:D
193	\name_primitive:NN	\ifvoid	\tex_ifvoid:D
194	\name_primitive:NN	\ifx	\tex_ifx:D
195	\name_primitive:NN	\iffalse	\tex_iffalse:D
196	\name_primitive:NN	\iftrue	\tex_iftrue:D
197	\name_primitive:NN	\ifhmode	\tex_ifhmode:D
198	\name_primitive:NN	\ifmmode	\tex_ifmmode:D
199	\name_primitive:NN	\ifvmode	\tex_ifvmode:D
200	\name_primitive:NN	\ifinner	\tex_ifinner:D
201	\name_primitive:NN	\else	\tex_else:D
202	\name_primitive:NN	\fi	\tex_fi:D
203	\name_primitive:NN	\or	\tex_or:D
204	\name_primitive:NN	\immediate	\tex_immediate:D
205	\name_primitive:NN	\closeout	\tex_closeout:D
206	\name_primitive:NN	\openin	\tex_openin:D
207	\name_primitive:NN	\openout	\tex_openout:D

208	\name_primitive:NN \read	\tex_read:D
209	\name_primitive:NN \write	\tex_write:D
210	\name_primitive:NN \closein	\tex_closein:D
211	\name_primitive:NN \newlinechar	\tex_newlinechar:D
212	\name_primitive:NN \input	\tex_input:D
213	\name_primitive:NN \endinput	\tex_endinput:D
214	\name_primitive:NN \inputlineno	\tex_inputlineno:D
215	\name_primitive:NN \errmessage	\tex_errmessage:D
216	\name_primitive:NN \message	\tex_message:D
217	\name_primitive:NN \show	\tex_show:D
218	\name_primitive:NN \showthe	\tex_showthe:D
219	\name_primitive:NN \showbox	\tex_showbox:D
220	\name_primitive:NN \showlists	\tex_showlists:D
221	\name_primitive:NN \errhelp	\tex_errhelp:D
222	\name_primitive:NN \errorcontextlines	\tex_errorcontextlines:D
223	\name_primitive:NN \tracingcommands	\tex_tracingcommands:D
224	\name_primitive:NN \tracinglostchars	\tex_tracinglostchars:D
225	\name_primitive:NN \tracingmacros	\tex_tracingmacros:D
226	\name_primitive:NN \tracingonline	\tex_tracingonline:D
227	\name_primitive:NN \tracingoutput	\tex_tracingoutput:D
228	\name_primitive:NN \tracingpages	\tex_tracingpages:D
229	\name_primitive:NN \tracingparagraphs	\tex_tracingparagraphs:D
230	\name_primitive:NN \tracingrestores	\tex_tracingrestores:D
231	\name_primitive:NN \tracingstats	\tex_tracingstats:D
232	\name_primitive:NN \pausing	\tex_pausing:D
233	\name_primitive:NN \showboxbreadth	\tex_showboxbreadth:D
234	\name_primitive:NN \showboxdepth	\tex_showboxdepth:D
235	\name_primitive:NN \batchmode	\tex_batchmode:D
236	\name_primitive:NN \errorstopmode	\tex_errorstopmode:D
237	\name_primitive:NN \nonstopmode	\tex_nonstopmode:D
238	\name_primitive:NN \scrollmode	\tex_scrollmode:D
239	\name_primitive:NN \end	\tex_end:D
240	\name_primitive:NN \csname	\tex_csname:D
241	\name_primitive:NN \endcsname	\tex_endcsname:D
242	\name_primitive:NN \ignorespaces	\tex_ignorespaces:D
243	\name_primitive:NN \relax	\tex_relax:D
244	\name_primitive:NN \the	\tex_the:D
245	\name_primitive:NN \mag	\tex_mag:D
246	\name_primitive:NN \language	\tex_language:D
247	\name_primitive:NN \mark	\tex_mark:D
248	\name_primitive:NN \topmark	\tex_topmark:D
249	\name_primitive:NN \firstmark	\tex_firstmark:D
250	\name_primitive:NN \botmark	\tex_botmark:D
251	\name_primitive:NN \splitfirstmark	\tex_splitfirstmark:D
252	\name_primitive:NN \splitbotmark	\tex_splitbotmark:D
253	\name_primitive:NN \fontname	\tex_fontname:D
254	\name_primitive:NN \escapechar	\tex_escapechar:D
255	\name_primitive:NN \endlinechar	\tex_endlinechar:D
256	\name_primitive:NN \mathchoice	\tex_mathchoice:D
257	\name_primitive:NN \delimiter	\tex_delimiter:D

258	\name_primitive:NN	\mathaccent	\tex_mathaccent:D
259	\name_primitive:NN	\mathchar	\tex_mathchar:D
260	\name_primitive:NN	\mskip	\tex_mskip:D
261	\name_primitive:NN	\radical	\tex_radical:D
262	\name_primitive:NN	\vcenter	\tex_vcenter:D
263	\name_primitive:NN	\mkern	\tex_mkern:D
264	\name_primitive:NN	\above	\tex_above:D
265	\name_primitive:NN	\abovewithdelims	\tex_abovewithdelims:D
266	\name_primitive:NN	\atop	\tex_atop:D
267	\name_primitive:NN	\atopwithdelims	\tex_atopwithdelims:D
268	\name_primitive:NN	\over	\tex_over:D
269	\name_primitive:NN	\overwithdelims	\tex_overwithdelims:D
270	\name_primitive:NN	\displaystyle	\tex_displaystyle:D
271	\name_primitive:NN	\textstyle	\tex_textstyle:D
272	\name_primitive:NN	\scriptstyle	\tex_scriptstyle:D
273	\name_primitive:NN	\scriptscriptstyle	\tex_scriptscriptstyle:D
274	\name_primitive:NN	\nonscript	\tex_nonscript:D
275	\name_primitive:NN	\eqno	\tex_eqno:D
276	\name_primitive:NN	\leqno	\tex_leqno:D
277	\name_primitive:NN	\abovedisplayshortskip	\tex_abovedisplayshortskip:D
278	\name_primitive:NN	\abovedisplayskip	\tex_abovedisplayskip:D
279	\name_primitive:NN	\belowdisplayshortskip	\tex_belowdisplayshortskip:D
280	\name_primitive:NN	\belowdisplayskip	\tex_belowdisplayskip:D
281	\name_primitive:NN	\displaywidowpenalty	\tex_displaywidowpenalty:D
282	\name_primitive:NN	\displayindent	\tex_displayindent:D
283	\name_primitive:NN	\displaywidth	\tex_displaywidth:D
284	\name_primitive:NN	\everydisplay	\tex_everydisplay:D
285	\name_primitive:NN	\predisplaysize	\tex_predisplaysize:D
286	\name_primitive:NN	\predisdisplaypenalty	\tex_predisdisplaypenalty:D
287	\name_primitive:NN	\postdisplaypenalty	\tex_postdisplaypenalty:D
288	\name_primitive:NN	\mathbin	\tex_mathbin:D
289	\name_primitive:NN	\mathclose	\tex_mathclose:D
290	\name_primitive:NN	\mathinner	\tex_mathinner:D
291	\name_primitive:NN	\mathop	\tex_mathop:D
292	\name_primitive:NN	\displaylimits	\tex_displaylimits:D
293	\name_primitive:NN	\limits	\tex_limits:D
294	\name_primitive:NN	\nolimits	\tex_nolimits:D
295	\name_primitive:NN	\mathopen	\tex_mathopen:D
296	\name_primitive:NN	\mathord	\tex_mathord:D
297	\name_primitive:NN	\mathpunct	\tex_mathpunct:D
298	\name_primitive:NN	\mathrel	\tex_mathrel:D
299	\name_primitive:NN	\overline	\tex_overline:D
300	\name_primitive:NN	\underline	\tex_underline:D
301	\name_primitive:NN	\left	\tex_left:D
302	\name_primitive:NN	\right	\tex_right:D
303	\name_primitive:NN	\binoppenalty	\tex_binoppenalty:D
304	\name_primitive:NN	\relpenalty	\tex_relpenalty:D
305	\name_primitive:NN	\delimitershortfall	\tex_delimitershortfall:D
306	\name_primitive:NN	\delimiterfactor	\tex_delimiterfactor:D
307	\name_primitive:NN	\nulldelimiterspace	\tex_nulldelimiterspace:D

308	<code>\name_primitive:NN \everymath</code>	<code>\tex_everymath:D</code>
309	<code>\name_primitive:NN \mathsurround</code>	<code>\tex_mathsurround:D</code>
310	<code>\name_primitive:NN \medmuskip</code>	<code>\tex_medmuskip:D</code>
311	<code>\name_primitive:NN \thinmuskip</code>	<code>\tex_thinmuskip:D</code>
312	<code>\name_primitive:NN \thickmuskip</code>	<code>\tex_thickmuskip:D</code>
313	<code>\name_primitive:NN \scriptspace</code>	<code>\tex_scriptspace:D</code>
314	<code>\name_primitive:NN \noboundary</code>	<code>\tex_noboundary:D</code>
315	<code>\name_primitive:NN \accent</code>	<code>\tex_accent:D</code>
316	<code>\name_primitive:NN \char</code>	<code>\tex_char:D</code>
317	<code>\name_primitive:NN \discretionary</code>	<code>\tex_discretionary:D</code>
318	<code>\name_primitive:NN \hfil</code>	<code>\tex_hfil:D</code>
319	<code>\name_primitive:NN \hfilneg</code>	<code>\tex_hfilneg:D</code>
320	<code>\name_primitive:NN \hfill</code>	<code>\tex_hfill:D</code>
321	<code>\name_primitive:NN \hskip</code>	<code>\tex_hskip:D</code>
322	<code>\name_primitive:NN \hss</code>	<code>\tex_hss:D</code>
323	<code>\name_primitive:NN \vfil</code>	<code>\tex_vfil:D</code>
324	<code>\name_primitive:NN \vfilneg</code>	<code>\tex_vfilneg:D</code>
325	<code>\name_primitive:NN \vfill</code>	<code>\tex_vfill:D</code>
326	<code>\name_primitive:NN \vskip</code>	<code>\tex_vskip:D</code>
327	<code>\name_primitive:NN \vss</code>	<code>\tex_vss:D</code>
328	<code>\name_primitive:NN \unskip</code>	<code>\tex_unskip:D</code>
329	<code>\name_primitive:NN \kern</code>	<code>\tex_kern:D</code>
330	<code>\name_primitive:NN \unkern</code>	<code>\tex_unkern:D</code>
331	<code>\name_primitive:NN \hrule</code>	<code>\tex_hrule:D</code>
332	<code>\name_primitive:NN \vrule</code>	<code>\tex_vrule:D</code>
333	<code>\name_primitive:NN \leaders</code>	<code>\tex_leaders:D</code>
334	<code>\name_primitive:NN \cleaders</code>	<code>\tex_cleaders:D</code>
335	<code>\name_primitive:NN \xleaders</code>	<code>\tex_xleaders:D</code>
336	<code>\name_primitive:NN \lastkern</code>	<code>\tex_lastkern:D</code>
337	<code>\name_primitive:NN \lastskip</code>	<code>\tex_lastskip:D</code>
338	<code>\name_primitive:NN \indent</code>	<code>\tex_indent:D</code>
339	<code>\name_primitive:NN \par</code>	<code>\tex_par:D</code>
340	<code>\name_primitive:NN \noindent</code>	<code>\tex_noindent:D</code>
341	<code>\name_primitive:NN \vadjust</code>	<code>\tex_vadjust:D</code>
342	<code>\name_primitive:NN \baselineskip</code>	<code>\tex_baselineskip:D</code>
343	<code>\name_primitive:NN \lineskip</code>	<code>\tex_lineskip:D</code>
344	<code>\name_primitive:NN \lineskiplimit</code>	<code>\tex_lineskiplimit:D</code>
345	<code>\name_primitive:NN \clubpenalty</code>	<code>\tex_clubpenalty:D</code>
346	<code>\name_primitive:NN \widowpenalty</code>	<code>\tex_widowpenalty:D</code>
347	<code>\name_primitive:NN \exhyphenpenalty</code>	<code>\tex_exhyphenpenalty:D</code>
348	<code>\name_primitive:NN \hyphenpenalty</code>	<code>\tex_hyphenpenalty:D</code>
349	<code>\name_primitive:NN \linepenalty</code>	<code>\tex_linepenalty:D</code>
350	<code>\name_primitive:NN \doublehyphendemerits</code>	<code>\tex_doublehyphendemerits:D</code>
351	<code>\name_primitive:NN \finalhyphendemerits</code>	<code>\tex_finalhyphendemerits:D</code>
352	<code>\name_primitive:NN \adjdemerits</code>	<code>\tex_adjdemerits:D</code>
353	<code>\name_primitive:NN \hangafter</code>	<code>\tex_hangafter:D</code>
354	<code>\name_primitive:NN \hangindent</code>	<code>\tex_hangindent:D</code>
355	<code>\name_primitive:NN \parshape</code>	<code>\tex_parshape:D</code>
356	<code>\name_primitive:NN \hsize</code>	<code>\tex_hsize:D</code>
357	<code>\name_primitive:NN \lefthyphenmin</code>	<code>\tex_lefthyphenmin:D</code>

358	\name_primitive:NN	\righthyphenmin	\tex_righthyphenmin:D
359	\name_primitive:NN	\leftskip	\tex_leftskip:D
360	\name_primitive:NN	\rightskip	\tex_rightskip:D
361	\name_primitive:NN	\looseness	\tex_looseness:D
362	\name_primitive:NN	\parskip	\tex_parskip:D
363	\name_primitive:NN	\parindent	\tex_parindent:D
364	\name_primitive:NN	\uchyph	\tex_uchyph:D
365	\name_primitive:NN	\emergencystretch	\tex_emergencystretch:D
366	\name_primitive:NN	\pretolerance	\tex_pretolerance:D
367	\name_primitive:NN	\tolerance	\tex_tolerance:D
368	\name_primitive:NN	\spaceskip	\tex_spaceskip:D
369	\name_primitive:NN	\xspaceskip	\tex_xspaceskip:D
370	\name_primitive:NN	\parfillskip	\tex_parfillskip:D
371	\name_primitive:NN	\everypar	\tex_everypar:D
372	\name_primitive:NN	\prevgraf	\tex_prevgraf:D
373	\name_primitive:NN	\spacefactor	\tex_spacefactor:D
374	\name_primitive:NN	\shipout	\tex_shipout:D
375	\name_primitive:NN	\vsize	\tex_vsize:D
376	\name_primitive:NN	\interlinepenalty	\tex_interlinepenalty:D
377	\name_primitive:NN	\brokenpenalty	\tex_brokenpenalty:D
378	\name_primitive:NN	\topskip	\tex_topskip:D
379	\name_primitive:NN	\maxdeadcycles	\tex_maxdeadcycles:D
380	\name_primitive:NN	\maxdepth	\tex_maxdepth:D
381	\name_primitive:NN	\output	\tex_output:D
382	\name_primitive:NN	\deadcycles	\tex_deadcycles:D
383	\name_primitive:NN	\pagedepth	\tex_pagedepth:D
384	\name_primitive:NN	\pagestretch	\tex_pagestretch:D
385	\name_primitive:NN	\pagefilstretch	\tex_pagefilstretch:D
386	\name_primitive:NN	\pagefillstretch	\tex_pagefillstretch:D
387	\name_primitive:NN	\pagefilllstretch	\tex_pagefilllstretch:D
388	\name_primitive:NN	\pageshrink	\tex_pageshrink:D
389	\name_primitive:NN	\pagegoal	\tex_pagegoal:D
390	\name_primitive:NN	\pagetotal	\tex_pagetotal:D
391	\name_primitive:NN	\outputpenalty	\tex_outputpenalty:D
392	\name_primitive:NN	\hoffset	\tex_hoffset:D
393	\name_primitive:NN	\voffset	\tex_voffset:D
394	\name_primitive:NN	\insert	\tex_insert:D
395	\name_primitive:NN	\holdinginserts	\tex_holdinginserts:D
396	\name_primitive:NN	\floatingpenalty	\tex_floatingpenalty:D
397	\name_primitive:NN	\insertpenalties	\tex_insertpenalties:D
398	\name_primitive:NN	\lower	\tex_lower:D
399	\name_primitive:NN	\moveleft	\tex_moveleft:D
400	\name_primitive:NN	\moveright	\tex_moveright:D
401	\name_primitive:NN	\raise	\tex_raise:D
402	\name_primitive:NN	\copy	\tex_copy:D
403	\name_primitive:NN	\lastbox	\tex_lastbox:D
404	\name_primitive:NN	\vsplit	\tex_vsplit:D
405	\name_primitive:NN	\unhbox	\tex_unhbox:D
406	\name_primitive:NN	\unhcopy	\tex_unhcopy:D
407	\name_primitive:NN	\unvbox	\tex_unvbox:D

408	\name_primitive:NN	\unvcopy	\tex_unvcopy:D
409	\name_primitive:NN	\setbox	\tex_setbox:D
410	\name_primitive:NN	\hbox	\tex_hbox:D
411	\name_primitive:NN	\vbox	\tex_vbox:D
412	\name_primitive:NN	\vtop	\tex_vtop:D
413	\name_primitive:NN	\prevdepth	\tex_prevdepth:D
414	\name_primitive:NN	\badness	\tex_badness:D
415	\name_primitive:NN	\hbadness	\tex_hbadness:D
416	\name_primitive:NN	\vbadness	\tex_vbadness:D
417	\name_primitive:NN	\hfuzz	\tex_hfuzz:D
418	\name_primitive:NN	\vfuzz	\tex_vfuzz:D
419	\name_primitive:NN	\overfullrule	\tex_overfullrule:D
420	\name_primitive:NN	\boxmaxdepth	\tex_boxmaxdepth:D
421	\name_primitive:NN	\splitmaxdepth	\tex_splitmaxdepth:D
422	\name_primitive:NN	\splittopskip	\tex_splittopskip:D
423	\name_primitive:NN	\everyhbox	\tex_everyhbox:D
424	\name_primitive:NN	\everyvbox	\tex_everyvbox:D
425	\name_primitive:NN	\nullfont	\tex_nullfont:D
426	\name_primitive:NN	\textfont	\tex_textfont:D
427	\name_primitive:NN	\scriptfont	\tex_scriptfont:D
428	\name_primitive:NN	\scriptscriptfont	\tex_scriptscriptfont:D
429	\name_primitive:NN	\fontdimen	\tex_fontdimen:D
430	\name_primitive:NN	\hyphenchar	\tex_hyphenchar:D
431	\name_primitive:NN	\skewchar	\tex_skewchar:D
432	\name_primitive:NN	\defaultshyphenchar	\tex_defaultshyphenchar:D
433	\name_primitive:NN	\defaultskewchar	\tex_defaultskewchar:D
434	\name_primitive:NN	\number	\tex_number:D
435	\name_primitive:NN	\romannumeral	\tex_romannumeral:D
436	\name_primitive:NN	\string	\tex_string:D
437	\name_primitive:NN	\lowercase	\tex_lowercase:D
438	\name_primitive:NN	\uppercase	\tex_uppercase:D
439	\name_primitive:NN	\meaning	\tex_meaning:D
440	\name_primitive:NN	\penalty	\tex_penalty:D
441	\name_primitive:NN	\unpenalty	\tex_unpenalty:D
442	\name_primitive:NN	\lastpenalty	\tex_lastpenalty:D
443	\name_primitive:NN	\special	\tex_special:D
444	\name_primitive:NN	\dump	\tex_dump:D
445	\name_primitive:NN	\patterns	\tex_patterns:D
446	\name_primitive:NN	\hyphenation	\tex_hyphenation:D
447	\name_primitive:NN	\time	\tex_time:D
448	\name_primitive:NN	\day	\tex_day:D
449	\name_primitive:NN	\month	\tex_month:D
450	\name_primitive:NN	\year	\tex_year:D
451	\name_primitive:NN	\jobname	\tex_jobname:D
452	\name_primitive:NN	\everyjob	\tex_everyjob:D
453	\name_primitive:NN	\count	\tex_count:D
454	\name_primitive:NN	\dimen	\tex_dimen:D
455	\name_primitive:NN	\skip	\tex_skip:D
456	\name_primitive:NN	\toks	\tex_toks:D
457	\name_primitive:NN	\muskip	\tex_muskip:D

458	<code>\name_primitive:NN \box</code>	<code>\tex_box:D</code>
459	<code>\name_primitive:NN \wd</code>	<code>\tex_wd:D</code>
460	<code>\name_primitive:NN \ht</code>	<code>\tex_ht:D</code>
461	<code>\name_primitive:NN \dp</code>	<code>\tex_dp:D</code>
462	<code>\name_primitive:NN \catcode</code>	<code>\tex_catcode:D</code>
463	<code>\name_primitive:NN \delcode</code>	<code>\tex_delcode:D</code>
464	<code>\name_primitive:NN \sfcode</code>	<code>\tex_sfcode:D</code>
465	<code>\name_primitive:NN \lccode</code>	<code>\tex_lccode:D</code>
466	<code>\name_primitive:NN \uccode</code>	<code>\tex_uccode:D</code>
467	<code>\name_primitive:NN \mathcode</code>	<code>\tex_mathcode:D</code>

Since L^AT_EX3 requires at least the ε -T_EX extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

468	<code>\name_primitive:NN \ifdefined</code>	<code>\etex_ifdefined:D</code>
469	<code>\name_primitive:NN \ifcsname</code>	<code>\etex_ifcsname:D</code>
470	<code>\name_primitive:NN \unless</code>	<code>\etex_unless:D</code>
471	<code>\name_primitive:NN \eTeXversion</code>	<code>\etex_eTeXversion:D</code>
472	<code>\name_primitive:NN \eTeXrevision</code>	<code>\etex_eTeXrevision:D</code>
473	<code>\name_primitive:NN \marks</code>	<code>\etex_marks:D</code>
474	<code>\name_primitive:NN \topmarks</code>	<code>\etex_topmarks:D</code>
475	<code>\name_primitive:NN \firstmarks</code>	<code>\etex_firstmarks:D</code>
476	<code>\name_primitive:NN \botmarks</code>	<code>\etex_botmarks:D</code>
477	<code>\name_primitive:NN \splitfirstmarks</code>	<code>\etex_splitfirstmarks:D</code>
478	<code>\name_primitive:NN \splitbotmarks</code>	<code>\etex_splitbotmarks:D</code>
479	<code>\name_primitive:NN \unexpanded</code>	<code>\etex_unexpanded:D</code>
480	<code>\name_primitive:NN \detokenize</code>	<code>\etex_detokenize:D</code>
481	<code>\name_primitive:NN \scantokens</code>	<code>\etex_scantokens:D</code>
482	<code>\name_primitive:NN \showtokens</code>	<code>\etex_showtokens:D</code>
483	<code>\name_primitive:NN \readline</code>	<code>\etex_readline:D</code>
484	<code>\name_primitive:NN \tracingassigns</code>	<code>\etex_tracingassigns:D</code>
485	<code>\name_primitive:NN \tracingscantokens</code>	<code>\etex_tracingscantokens:D</code>
486	<code>\name_primitive:NN \tracingnesting</code>	<code>\etex_tracingnesting:D</code>
487	<code>\name_primitive:NN \tracingifs</code>	<code>\etex_tracingifs:D</code>
488	<code>\name_primitive:NN \currentiflevel</code>	<code>\etex_currentiflevel:D</code>
489	<code>\name_primitive:NN \currentifbranch</code>	<code>\etex_currentifbranch:D</code>
490	<code>\name_primitive:NN \currentifttype</code>	<code>\etex_currentifttype:D</code>
491	<code>\name_primitive:NN \tracinggroups</code>	<code>\etex_tracinggroups:D</code>
492	<code>\name_primitive:NN \currentgrouplevel</code>	<code>\etex_currentgrouplevel:D</code>
493	<code>\name_primitive:NN \currentgrouptype</code>	<code>\etex_currentgrouptype:D</code>
494	<code>\name_primitive:NN \showgroups</code>	<code>\etex_showgroups:D</code>
495	<code>\name_primitive:NN \showifs</code>	<code>\etex_showifs:D</code>
496	<code>\name_primitive:NN \interactionmode</code>	<code>\etex_interactionmode:D</code>
497	<code>\name_primitive:NN \lastnodetype</code>	<code>\etex_lastnodetype:D</code>
498	<code>\name_primitive:NN \iffontchar</code>	<code>\etex_iffontchar:D</code>
499	<code>\name_primitive:NN \fontcharht</code>	<code>\etex_fontcharht:D</code>
500	<code>\name_primitive:NN \fontchardp</code>	<code>\etex_fontchardp:D</code>
501	<code>\name_primitive:NN \fontcharwd</code>	<code>\etex_fontcharwd:D</code>
502	<code>\name_primitive:NN \fontcharic</code>	<code>\etex_fontcharic:D</code>
503	<code>\name_primitive:NN \parshapeindent</code>	<code>\etex_parshapeindent:D</code>

504	\name_primitive:NN	\parshapelength	\etex_parshapelength:D
505	\name_primitive:NN	\parshapedimen	\etex_parshapedimen:D
506	\name_primitive:NN	\numexpr	\etex_numexpr:D
507	\name_primitive:NN	\dimexpr	\etex_dimexpr:D
508	\name_primitive:NN	\glueexpr	\etex_glueexpr:D
509	\name_primitive:NN	\muexpr	\etex_muexpr:D
510	\name_primitive:NN	\gluestretch	\etex_gluestretch:D
511	\name_primitive:NN	\glueshrink	\etex_glueshrink:D
512	\name_primitive:NN	\gluestretchorder	\etex_gluestretchorder:D
513	\name_primitive:NN	\glueshrinkorder	\etex_glueshrinkorder:D
514	\name_primitive:NN	\gluetomu	\etex_gluetomu:D
515	\name_primitive:NN	\mutoglua	\etex_mutoglua:D
516	\name_primitive:NN	\lastlinefit	\etex_lastlinefit:D
517	\name_primitive:NN	\interlinepenalties	\etex_interlinepenalties:D
518	\name_primitive:NN	\clubpenalties	\etex_clubpenalties:D
519	\name_primitive:NN	\widowpenalties	\etex_widowpenalties:D
520	\name_primitive:NN	\displaywidowpenalties	\etex_displaywidowpenalties:D
521	\name_primitive:NN	\middle	\etex_middle:D
522	\name_primitive:NN	\savinghyphcodes	\etex_savinghyphcodes:D
523	\name_primitive:NN	\savingvdiscards	\etex_savingvdiscards:D
524	\name_primitive:NN	\pagediscards	\etex_pagediscards:D
525	\name_primitive:NN	\splitdiscards	\etex_splitdiscards:D
526	\name_primitive:NN	\TeXETstate	\etex_TeXETstate:D
527	\name_primitive:NN	\beginL	\etex_beginL:D
528	\name_primitive:NN	\endL	\etex_endL:D
529	\name_primitive:NN	\beginR	\etex_beginR:D
530	\name_primitive:NN	\endR	\etex_endR:D
531	\name_primitive:NN	\predisplaydirection	\etex_predisplaydirection:D
532	\name_primitive:NN	\everyeof	\etex_everyeof:D
533	\name_primitive:NN	\protected	\etex_protected:D

All major distributions use pdf ϵ -T ϵ X as engine so we add these names as well. Since the pdfT ϵ X team has been very good at prefixing most primitives with pdf (so far only five do not start with pdf) we do not give then a double pdf prefix. The list below covers pdfT ϵ Xv 1.30.4.

534	%% integer registers:		
535	\name_primitive:NN	\pdfoutput	\pdf_output:D
536	\name_primitive:NN	\pdfminorversion	\pdf_minorversion:D
537	\name_primitive:NN	\pdfcompresslevel	\pdf_compresslevel:D
538	\name_primitive:NN	\pdfdecimaldigits	\pdf_decimaldigits:D
539	\name_primitive:NN	\pdfimageresolution	\pdf_imageresolution:D
540	\name_primitive:NN	\pdfpkresolution	\pdf_pkresolution:D
541	\name_primitive:NN	\pdftracingfonts	\pdf_tracingfonts:D
542	\name_primitive:NN	\pdfuniqueresname	\pdf_uniqueresname:D
543	\name_primitive:NN	\pdfadjustspacing	\pdf_adjustspacing:D
544	\name_primitive:NN	\pdfprotrudechars	\pdf_protrudechars:D
545	\name_primitive:NN	\efcode	\pdf_efcode:D
546	\name_primitive:NN	\lpcode	\pdf_lpcode:D
547	\name_primitive:NN	\rpcode	\pdf_rpcode:D

```

548 \name_primitive:NN \pdfforcepagebox \pdf_forcepagebox:D
549 \name_primitive:NN \pdfoptionalwaysusepdfpagebox \pdf_optionalwaysusepdfpagebox:D
550 \name_primitive:NN \pdfinclusionerrorlevel\pdf_inclusionerrorlevel:D
551 \name_primitive:NN \pdfoptionpdfinclusionerrorlevel \pdf_optionpdfinclusionerrorlevel:D
552 \name_primitive:NN \pdfimagehicolor \pdf_imagehicolor:D
553 \name_primitive:NN \pdfimageapplygamma \pdf_imageapplygamma:D
554 \name_primitive:NN \pdfgamma \pdf_gamma:D
555 \name_primitive:NN \pdfimagegamma \pdf_imagegamma:D
556 %% dimen registers:
557 \name_primitive:NN \pdfhorigin \pdf_horigin:D
558 \name_primitive:NN \pdfvorigin \pdf_vorigin:D
559 \name_primitive:NN \pdfpagewidth \pdf_pagewidth:D
560 \name_primitive:NN \pdfpageheight \pdf_pageheight:D
561 \name_primitive:NN \pdflinkmargin \pdf_linkmargin:D
562 \name_primitive:NN \pdfdestmargin \pdf_destmargin:D
563 \name_primitive:NN \pdfthreadmargin \pdf_threadmargin:D
564 %% token registers:
565 \name_primitive:NN \pdfpagesattr \pdf_pagesattr:D
566 \name_primitive:NN \pdfpageattr \pdf_pageattr:D
567 \name_primitive:NN \pdfpageresources \pdf_pageresources:D
568 \name_primitive:NN \pdfpkmode \pdf_pkmode:D
569 %% expandable commands:
570 \name_primitive:NN \pdftexrevision \pdf_texrevision:D
571 \name_primitive:NN \pdftexbanner \pdf_texbanner:D
572 \name_primitive:NN \pdfcreationdate \pdf_creationdate:D
573 \name_primitive:NN \pdfpageref \pdf_pageref:D
574 \name_primitive:NN \pdfxformname \pdf_xformname:D
575 \name_primitive:NN \pdffontname \pdf_fontname:D
576 \name_primitive:NN \pdffontobjnum \pdf_fontobjnum:D
577 \name_primitive:NN \pdffontsize \pdf_fontsize:D
578 \name_primitive:NN \pdfincludechars \pdf_includechars:D
579 \name_primitive:NN \leftmarginkern \pdf_leftmarginkern:D
580 \name_primitive:NN \rightmarginkern \pdf_rightmarginkern:D
581 \name_primitive:NN \pdfescapestring \pdf_escapestring:D
582 \name_primitive:NN \pdfescapename \pdf_escapename:D
583 \name_primitive:NN \pdfescapehex \pdf_escapehex:D
584 \name_primitive:NN \pdfunescapehex \pdf_unescapehex:D
585 \name_primitive:NN \pdfstrcmp \pdf_strcmp:D
586 \name_primitive:NN \pdfuniformdeviate \pdf_uniformdeviate:D
587 \name_primitive:NN \pdfnormaldeviate \pdf_normaldeviate:D
588 \name_primitive:NN \pdfmdfivesum \pdf_mdfivesum:D
589 \name_primitive:NN \pdffilemoddate \pdf_filemoddate:D
590 \name_primitive:NN \pdffilesize \pdf_filesize:D
591 \name_primitive:NN \pdffiledump \pdf_filedump:D
592 %% read-only integers:
593 \name_primitive:NN \pdftexversion \pdf_texversion:D
594 \name_primitive:NN \pdflastobj \pdf_lastobj:D
595 \name_primitive:NN \pdflastxform \pdf_lastxform:D
596 \name_primitive:NN \pdflastximage \pdf_lastximage:D
597 \name_primitive:NN \pdflastximagepages \pdf_lastximagepages:D

```

```

598 \name_primitive:NN \pdflastannot          \pdf_lastannot:D
599 \name_primitive:NN \pdflastxpos           \pdf_lastxpos:D
600 \name_primitive:NN \pdflastypos           \pdf_lastypos:D
601 \name_primitive:NN \pdflastdemerits       \pdf_lastdemerits:D
602 \name_primitive:NN \pdfelapsedtime        \pdf_elapsedtime:D
603 \name_primitive:NN \pdfrandomseed         \pdf_randomseed:D
604 \name_primitive:NN \pdfshellescape        \pdf_shellescape:D
605 %% general commands:
606 \name_primitive:NN \pdfobj                 \pdf_obj:D
607 \name_primitive:NN \pdfrefobj             \pdf_refobj:D
608 \name_primitive:NN \pdfxform              \pdf_xform:D
609 \name_primitive:NN \pdfrefxform           \pdf_refxform:D
610 \name_primitive:NN \pdfximage             \pdf_ximage:D
611 \name_primitive:NN \pdfrefximage          \pdf_refximage:D
612 \name_primitive:NN \pdfannot             \pdf_annot:D
613 \name_primitive:NN \pdfstartlink          \pdf_startlink:D
614 \name_primitive:NN \pdfendlink            \pdf_endlink:D
615 \name_primitive:NN \pdfoutline            \pdf_outline:D
616 \name_primitive:NN \pdfdest              \pdf_dest:D
617 \name_primitive:NN \pdfthread             \pdf_thread:D
618 \name_primitive:NN \pdfstartthread        \pdf_startthread:D
619 \name_primitive:NN \pdfendthread          \pdf_endthread:D
620 \name_primitive:NN \pdfsavepos            \pdf_savepos:D
621 \name_primitive:NN \pdfinfo               \pdf_info:D
622 \name_primitive:NN \pdfcatalog            \pdf_catalog:D
623 \name_primitive:NN \pdfnames              \pdf_names:D
624 \name_primitive:NN \pdfmapfile            \pdf_mapfile:D
625 \name_primitive:NN \pdfmapline            \pdf_mapline:D
626 \name_primitive:NN \pdffontattr           \pdf_fontattr:D
627 \name_primitive:NN \pdftrailer            \pdf_trailer:D
628 \name_primitive:NN \pdffontexpand         \pdf_fontexpand:D
629 %%\name_primitive:NN \adjust [<pre spec>] <filler> { <vertical mode material> } (h, m)
630 \name_primitive:NN \pdfliteral            \pdf_literal:D
631 %%\name_primitive:NN \special <pdfspecial spec>
632 \name_primitive:NN \pdfresettimer         \pdf_resettimer:D
633 \name_primitive:NN \pdfsetrandomseed      \pdf_setrandomseed:D
634 \name_primitive:NN \pdfnoligatures        \pdf_noligatures:D

```

Only a little bit of XeTeX and LuaTeX at the moment.

```

635 \name_primitive:NN \XeTeXversion          \xetex_version:D
636 \name_primitive:NN \catcodetable          \luatex_catcodetable:D
637 \name_primitive:NN \directlua             \luatex_directlua:D
638 \name_primitive:NN \initcatcodetable       \luatex_initcatcodetable:D
639 \name_primitive:NN \latelua               \luatex_latelua:D
640 \name_primitive:NN \savecatcodetable       \luatex_savecatcodetable:D

```

XeTeX adds `\strcmp` to the set of primitives, with the same implementation as `\pdfstrcmp` but a different name. To avoid having to worry about this later, the same internal name is used. Note that we want to use the original primitive name here so that

`\name_undefine:N` (if used) will remove it.

```

641 \tex_begingroup:D
642   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
643 \tex_endgroup:D
644 \tex_expandafter:D \tex_ifx:D \tex_csname:D xetex_version:D\tex_endcsname:D
645   \tex_relax:D \tex_else:D
646   \name_primitive:NN \strcmp \pdf_strcmp:D
647 \tex_fi:D

```

94.7 expl3 code switches

`\ExplSyntaxOn`
`\ExplSyntaxOff`
`\ExplSyntaxStatus`

Here we define functions that are used to turn on and off the special conventions used in the kernel of L^AT_EX3.

First of all, the space, tab and the return characters will all be ignored inside L^AT_EX3 code, the latter because `endline` is set to a space instead. When space characters are needed in L^AT_EX3 code the `~` character will be used for that purpose.

Specification of the desired behavior:

- `ExplSyntax` can be either On or Off.
- The On switch is `<null>` if `ExplSyntax` is on.
- The Off switch is `<null>` if `ExplSyntax` is off.
- If the On switch is issued and not `<null>`, it records the current catcode scheme just prior to it being issued.
- An Off switch restores the catcode scheme to what it was just prior to the previous On switch.

```

648 \etex_protected:D \tex_def:D \ExplSyntaxOn {
649   \tex_ifodd:D \ExplSyntaxStatus \tex_relax:D
650   \tex_else:D
651     \etex_protected:D \tex_edef:D \ExplSyntaxOff {
652       \etex_unexpanded:D{
653         \tex_ifodd:D \ExplSyntaxStatus \tex_relax:D
654         \tex_def:D \ExplSyntaxStatus{0}
655       }
656       \tex_catcode:D 126=\tex_the:D \tex_catcode:D 126 \tex_relax:D
657       \tex_catcode:D 32=\tex_the:D \tex_catcode:D 32 \tex_relax:D
658       \tex_catcode:D 9=\tex_the:D \tex_catcode:D 9 \tex_relax:D
659       \tex_endlinechar:D =\tex_the:D \tex_endlinechar:D \tex_relax:D
660       \tex_catcode:D 95=\tex_the:D \tex_catcode:D 95 \tex_relax:D
661       \tex_catcode:D 58=\tex_the:D \tex_catcode:D 58 \tex_relax:D
662       \tex_catcode:D 124=\tex_the:D \tex_catcode:D 124 \tex_relax:D
663       \tex_catcode:D 38=\tex_the:D \tex_catcode:D 38 \tex_relax:D
664       \tex_catcode:D 94=\tex_the:D \tex_catcode:D 94 \tex_relax:D

```

```

665     \tex_catcode:D 34=\tex_the:D \tex_catcode:D 34 \tex_relax:D
666     \tex_noexpand:D \tex_fi:D
667   }
668   \tex_def:D \ExplSyntaxStatus { 1 }
669   \tex_catcode:D 126=10 \tex_relax:D % tilde is a space char.
670   \tex_catcode:D 32=9 \tex_relax:D % space is ignored
671   \tex_catcode:D 9=9 \tex_relax:D % tab also ignored
672   \tex_endlinechar:D =32 \tex_relax:D % endline is space
673   \tex_catcode:D 95=11 \tex_relax:D % underscore letter
674   \tex_catcode:D 58=11 \tex_relax:D % colon letter
675   \tex_catcode:D 124=12 \tex_relax:D % vertical bar, other
676   \tex_catcode:D 38=4 \tex_relax:D % ampersand, alignment token
677   \tex_catcode:D 94=7 \tex_relax:D % caret, math superscript
678   \tex_catcode:D 34=12 \tex_relax:D % doublequote, other
679   \tex_fi:D
680 }

```

At this point we better set the status.

```

681 \tex_def:D \ExplSyntaxStatus { 1 }

```

(End definition for `\ExplSyntaxOn`. This function is documented on page 159.)

`\ExplSyntaxNamesOn`
`\ExplSyntaxNamesOff`

Sometimes we need to be able to use names from the kernel of L^AT_EX3 without adhering it's conventions according to space characters. These macros provide the necessary settings.

```

682 \etex_protected:D \tex_def:D \ExplSyntaxNamesOn {
683   \tex_catcode:D '\_ =11\tex_relax:D
684   \tex_catcode:D '\: =11\tex_relax:D
685 }
686 \etex_protected:D \tex_def:D \ExplSyntaxNamesOff {
687   \tex_catcode:D '\_ =8\tex_relax:D
688   \tex_catcode:D '\: =12\tex_relax:D
689 }

```

(End definition for `\ExplSyntaxNamesOn`. This function is documented on page 6.)

94.8 Package loading

`\GetIdInfo`
`\filedescription`
`\filename`
`\fileversion`
`\fileauthor`
`\filedate`
`\filenameext`
`\filetimestamp`
`\GetIdInfoAux:w`
`\GetIdInfoAuxii:w`
`\GetIdInfoAuxCVS:w`
`\GetIdInfoAuxSVN:w`

Extract all information from a cvs or svn field. The formats are slightly different but at least the information is in the same positions so we check in the date format so see if it contains a / after the four-digit year. If it does it is cvs else svn and we extract information. To be on the safe side we ensure that spaces in the argument are seen.

```

690 \etex_protected:D \tex_def:D \GetIdInfo {
691   \tex_begingroup:D
692   \tex_catcode:D 32=10 \tex_relax:D % needed? Probably for now.
693   \GetIdInfoMaybeMissing:w
694 }

```

```

695 \etex_protected:D \tex_def:D\GetIdInfoMaybeMissing:w$#1$#2{
696   \tex_def:D \l_kernel_tmpa_tl {#1}
697   \tex_def:D \l_kernel_tmpp_tl {Id}
698   \tex_ifx:D \l_kernel_tmpa_tl \l_kernel_tmpp_tl
699   \tex_def:D \l_kernel_tmpa_tl {
700     \tex_endgroup:D
701     \tex_def:D\filedescription{#2}
702     \tex_def:D\filename      {[unknown~name]}
703     \tex_def:D\fileversion   {000}
704     \tex_def:D\fileauthor    {[unknown~author]}
705     \tex_def:D\filedate      {0000/00/00}
706     \tex_def:D\filenameext   {[unknown~ext]}
707     \tex_def:D\filetimestamp {[unknown~timestamp]}
708   }
709   \tex_else:D
710     \tex_def:D \l_kernel_tmpa_tl {\GetIdInfoAuxii:w$#1$#2}}
711   \tex_fi:D
712   \l_kernel_tmpa_tl
713 }

714 \etex_protected:D \tex_def:D\GetIdInfoAuxii:w$#1~#2.#3~#4~#5~#6~#7~#8$#9{
715   \tex_endgroup:D
716   \tex_def:D\filename{#2}
717   \tex_def:D\fileversion{#4}
718   \tex_def:D\filedescription{#9}
719   \tex_def:D\fileauthor{#7}
720   \GetIdInfoAuxii:w #5\tex_relax:D
721   #3\tex_relax:D#5\tex_relax:D#6\tex_relax:D
722 }

723 \etex_protected:D \tex_def:D\GetIdInfoAuxii:w #1#2#3#4#5#6\tex_relax:D{
724   \tex_ifx:D#5/
725     \tex_expandafter:D\GetIdInfoAuxCVS:w
726   \tex_else:D
727     \tex_expandafter:D\GetIdInfoAuxSVN:w
728   \tex_fi:D
729 }

730 \etex_protected:D \tex_def:D\GetIdInfoAuxCVS:w #1,v\tex_relax:D
731   #2\tex_relax:D#3\tex_relax:D{
732   \tex_def:D\filedate{#2}
733   \tex_def:D\filenameext{#1}
734   \tex_def:D\filetimestamp{#3}

```

When creating the format we want the information in the log straight away.

```

735 <initex>\tex_immediate:D\tex_write:D-1
736 <initex> {\filename;~ v\fileversion,~\filedate;~\filedescription}
737 }
738 \etex_protected:D \tex_def:D\GetIdInfoAuxSVN:w #1\tex_relax:D#2~#3~#4

```

```

739 \tex_relax:D#5Z\tex_relax:D{
740 \tex_def:D\filenameext{#1}
741 \tex_def:D\filedate{#2/#3/#4}
742 \tex_def:D\filetimestamp{#5}
743 <-package>\tex_immediate:D\tex_write:D-1
744 <-package> {\filename;~ v\fileversion,~\filedate;~\filedescription}
745 }
746 </initex | package>

```

(End definition for `\GetIdInfo`. This function is documented on page 6.)

Finally some corrections in the case we are running over L^AT_EX 2_ε.

We want to set things up so that experimental packages and regular packages can coexist with the former using the L^AT_EX3 programming catcode settings. Since it cannot be the task of the end user to know how a package is constructed under the hood we make it so that the experimental packages have to identify themselves. As an example it can be done as

```

\RequirePackage{l3names}
\ProvidesExplPackage{agent}{2007/08/28}{007}{bonding module}

```

or by using the `\file{field}` informations from `\GetIdInfo` as the packages in this distribution do like this:

```

\RequirePackage{l3names}
\GetIdInfo$Id: l3names.dtx 2122 2011-01-08 09:14:28Z joseph $
    {L3 Experimental Box module}
\ProvidesExplPackage
    {\filename}{\filedate}{\fileversion}{\filedescription}

```

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

First up is the identification. Rather trivial as we don't allow for options just yet.

```

747 <*package>
748 \etex_protected:D \tex_def:D \ProvidesExplPackage#1#2#3#4{
749 \ProvidesPackage{#1}[#2~v#3~#4]
750 \ExplSyntaxOn
751 }
752 \etex_protected:D \tex_def:D \ProvidesExplClass#1#2#3#4{
753 \ProvidesClass{#1}[#2~v#3~#4]
754 \ExplSyntaxOn
755 }
756 \etex_protected:D \tex_def:D \ProvidesExplFile#1#2#3#4{
757 \ProvidesFile{#1}[#2~v#3~#4]
758 \ExplSyntaxOn
759 }

```

(End definition for `\ProvidesExplPackage`. This function is documented on page 6.)

\@pushfilename The idea behind the code is to record whether or not the L^AT_EX3 syntax is on or off when about to load a file with class or package extension. This status stored in the parameter **\ExplSyntaxStatus** and set by **\ExplSyntaxOn** and **\ExplSyntaxOff** to 1 and 0 respectively is pushed onto the stack **\ExplSyntaxStack**. Then the catcodes are set back to normal, the file loaded with its options and finally the stack is popped again. The whole thing is a bit problematical. So let's take a look at what the desired behavior is: A package or class which declares itself of Expl type by using **\ProvidesExplClass** or **\ProvidesExplPackage** should automatically ensure the correct catcode scheme as soon as the identification part is over. Similarly, a package or class which uses the traditional **\ProvidesClass** or **\ProvidesPackage** commands should go back to the traditional catcode scheme. An example:

```

\RequirePackage{l3names}
\ProvidesExplPackage{foobar}{2009/05/07}{0.1}{Foobar package}
\cs_new:Npn \foo_bar:nn #1#2 {#1,#2}
...
\RequirePackage{array}
...
\cs_new:Npn \foo_bar:nnn #1#2#3 {#3,#2,#1}

```

Inside the **array** package, everything should behave as normal under traditional L^AT_EX but as soon as we are back at the top level, we should use the new catcode regime.

Whenever L^AT_EX inputs a package file or similar, it calls upon **\@pushfilename** to push the name, the extension and the catcode of @ of the file it was currently processing onto a file name stack. Similarly, after inputting such a file, this file name stack is popped again and the catcode of @ is set to what it was before. If it is a package within package, @ maintains catcode 11 whereas if it is package within document preamble @ is reset to what it was in the preamble (which is usually catcode 12). We wish to adopt a similar technique. Every time an Expl package or class is declared, they will issue an **\ExplSyntaxOn**. Then whenever we are about to load another file, we will first push this status onto a stack and then turn it off again. Then when done loading a file, we pop the stack and if **\ExplSyntax** was On right before, so should it be now. The only problem with this is that we cannot guarantee that we get to the file name stack very early on. Therefore, if the **\ExplSyntaxStack** is empty when trying to pop it, we ensure to turn **\ExplSyntax** off again.

\@pushfilename is prepended with a small function pushing the current **\ExplSyntaxStatus** (true/false) onto a stack. Then the current catcode regime is recorded and **\ExplSyntax** is switched off.

\@popfilename is appended with a function for popping the **\ExplSyntax** stack. However, chances are we didn't get to hook into the file stack early enough so L^AT_EX might try to pop the file name stack while the **\ExplSyntaxStack** is empty. If the latter is empty, we just switch off **\ExplSyntax**.

```

760 \tex_edef:D \@pushfilename{

```

```

761 \etex_unexpanded:D{
762   \tex_edef:D \ExplSyntaxStack{ \ExplSyntaxStatus \ExplSyntaxStack }
763   \ExplSyntaxOff
764 }
765 \etex_unexpanded:D\tex_expandafter:D{\@pushfilename }
766 }
767 \tex_edef:D \@popfilename{
768   \etex_unexpanded:D\tex_expandafter:D{\@popfilename
769     \tex_if:D 2\ExplSyntaxStack 2
770     \ExplSyntaxOff
771     \tex_else:D
772     \tex_expandafter:D\ExplSyntaxPopStack\ExplSyntaxStack\q_stop
773     \tex_fi:D
774   }
775 }

```

(End definition for `\@pushfilename`. This function is documented on page 159.)

`\ExplSyntaxPopStack`
`\ExplSyntaxStack`

Popping the stack is simple: Take the first token which is either 0 (false) or 1 (true) and test if it is odd. Save the rest. The stack is initially empty set to 0 signalling that before `l3names` was loaded, the `ExplSyntax` was off.

```

776 \etex_protected:D\tex_def:D\ExplSyntaxPopStack#1#2\q_stop{
777   \tex_def:D\ExplSyntaxStack{#2}
778   \tex_ifodd:D#1\tex_relax:D
779   \ExplSyntaxOn
780   \tex_else:D
781   \ExplSyntaxOff
782   \tex_fi:D
783 }
784 \tex_def:D \ExplSyntaxStack{0}

```

(End definition for `\ExplSyntaxPopStack`. This function is documented on page 159.)

94.9 Finishing up

A few of the ‘primitives’ assigned above have already been stolen by L^AT_EX, so assign them by hand to the saved real primitive.

```

785 \tex_let:D\tex_input:D      \@@input
786 \tex_let:D\tex_underline:D  \@@underline
787 \tex_let:D\tex_end:D        \@@end
788 \tex_let:D\tex_everymath:D  \frozen@everymath
789 \tex_let:D\tex_everydisplay:D \frozen@everydisplay
790 \tex_let:D\tex_italiccorr:D \@@italiccorr
791 \tex_let:D\tex_hyphen:D      \@@hyph
792 \tex_let:D\luatex_catcodetable:D \luatexcatcodetable
793 \tex_let:D\luatex_initcatcodetable:D \luatexinitcatcodetable
794 \tex_let:D\luatex_latelua:D \luatexlatelua
795 \tex_let:D\luatex_savecatcodetable:D \luatexsavecatcodetable

```

TeX has a nasty habit of inserting a command with the name `\par` so we had better make sure that that command at least has a definition.

```
796 \tex_let:D\par \tex_par:D
```

This is the end for `l3names` when used on top of L^AT_EX 2_ε:

```
797 \tex_ifx:D\name_undefine:N\@gobble
798 \tex_def:D\name_pop_stack:w{}
799 \tex_else:D
```

But if traditional TeX code is disabled, do this...

As mentioned above, The L^AT_EX 2_ε package mechanism will insert some code to handle the filename stack, and reset the package options, this code will die if the TeX primitives have gone, so skip past it and insert some equivalent code that will work.

First a version of `\ProvidesPackage` that can cope.

```
800 \tex_def:D\ProvidesPackage{
801   \tex_begingroup:D
802   \ExplSyntaxOff
803   \package_provides:w}

804 \tex_def:D\package_provides:w#1#2[#3]{
805   \tex_endgroup:D
806   \tex_immediate:D\tex_write:D-1{Package:~#1#2~#3}
807   \tex_expandafter:D\tex_xdef:D
808   \tex_csname:D ver@#1.sty\tex_endcsname:D{#1}}
```

In this case the catcode preserving stack is not maintained and `\ExplSyntaxOn` conventions stay in force once on. You'll need to turn then off explicitly with `\ExplSyntaxOff` (although as currently built on 2e, nothing except very experimental code will run in this mode!) Also note that `\RequirePackage` is a simple definition, just for one file, with no options.

```
809 \tex_def:D\name_pop_stack:w#1\relax{%
810   \ExplSyntaxOff
811   \tex_expandafter:D\@p@pfilename\@currnamestack\@nil
812   \tex_let:D\default@ds\@unknownoptionerror
813   \tex_global:D\tex_let:D\ds@\@empty
814   \tex_global:D\tex_let:D\@declaredoptions\@empty}

815 \tex_def:D\@p@pfilename#1#2#3#4\@nil{%
816   \tex_gdef:D\@currname{#1}%
817   \tex_gdef:D\@currentx{#2}%
818   \tex_catcode:D'\@#3%
819   \tex_gdef:D\@currnamestack{#4}}
```

```

820 \tex_def:D\NeedsTeXFormat#1{}
821 \tex_def:D\RequirePackage#1{
822   \tex_expandafter:D\tex_ifx:D
823     \tex_csname:D ver@#1.sty\tex_endcsname:D\tex_relax:D
824     \ExplSyntaxOn
825     \tex_input:D#1.sty\tex_relax:D
826   \tex_fi:D}
827 \tex_fi:D

```

The `\futurelet` just forces the special end of file marker to vanish, so the argument of `\name_pop_stack:w` does not cause an end-of-file error. (Normally I use `\expandafter` for this trick, but here the next token is in fact `\let` and that may be undefined.)

```

828 \tex_futurelet:D\name_tmp:\name_pop_stack:w

```

expl3 dependency checks We want the `expl3` bundle to be loaded ‘as one’; this command is used to ensure that one of the 13 packages isn’t loaded on its own.

```

829 <*\initex>
830 \etex_protected:D\tex_def:D \package_check_loaded_expl: {
831   \@ifpackageloaded{expl3}{\{
832     \PackageError{expl3}{Cannot~load~the~expl3~modules~separately}{
833       The~expl3~modules~cannot~be~loaded~separately;\MessageBreak
834       please~\protect\usepackage{expl3}~instead.
835     }
836   }
837 }
838 </!\initex>
839 </package>

```

94.10 Showing memory usage

This section is from some old code from 1993; it’d be good to work out how it should be used in our code today.

During the development of the L^AT_EX3 kernel we need to be able to keep track of the memory usage. Therefore we generate empty pages while loading the kernel code, just to be able to check the memory usage.

```

840 <*\showmemory>
841 \g_trace_statistics_status=2\scan_stop:
842 \cs_set_nopar:Npn\showMemUsage{
843   \if_horizontal_mode:
844     \tex_errmessage:D{Wrong~ mode~ H:~ something~ triggered~
845       hmode~ above}
846   \else:
847     \tex_message:D{Mode ~ okay}

```

```

848 \fi:
849 \tex_shipout:D\hbox:w{}
850 }
851 \showMemUsage
852 </showmemory>

```

95 l3basics implementation

We need l3names to get things going but we actually need it very early on, so it is loaded at the very top of the file l3basics.dtx. Also, most of the code below won't run until l3expan has been loaded.

95.1 Renaming some T_EX primitives (again)

\cs_set_eq:NwN Having given all the tex primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but do a few now, just to get started.⁷

```

853 <*package>
854 \ProvidesExplPackage
855   {\filename}{\filedate}{\fileversion}{\filedescription}
856 \package_check_loaded_expl:
857 </package>
858 <*initex | package>
859 \tex_let:D \cs_set_eq:NwN          \tex_let:D

```

(End definition for \cs_set_eq:NwN. This function is documented on page 23.)

\if_true: Then some conditionals.
\if_false:
\or: 860 \cs_set_eq:NwN \if_true: \tex_iftrue:D
\else: 861 \cs_set_eq:NwN \if_false: \tex_iffalse:D
\fi: 862 \cs_set_eq:NwN \or: \tex_or:D
\reverse_if:N 863 \cs_set_eq:NwN \else: \tex_else:D
\if:w 864 \cs_set_eq:NwN \fi: \tex_fi:D
\if_bool:N 865 \cs_set_eq:NwN \reverse_if:N \etex_unless:D
\if_predicate:w 866 \cs_set_eq:NwN \if:w \tex_if:D
\if_charcode:w 867 \cs_set_eq:NwN \if_bool:N \tex_ifodd:D
\if_catcode:w 868 \cs_set_eq:NwN \if_predicate:w \tex_ifodd:D
869 \cs_set_eq:NwN \if_charcode:w \tex_if:D
870 \cs_set_eq:NwN \if_catcode:w \tex_ifcat:D

(End definition for \if_true:. This function is documented on page 9.)

⁷This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the “tex...D” name in the cases where no good alternative exists.

`\if_meaning:w`

```
871 \cs_set_eq:NwN \if_meaning:w \tex_ifx:D
```

(End definition for `\if_meaning:w`. This function is documented on page 9.)

`\if_mode_math:`

TeX lets us detect some of its modes.

`\if_mode_horizontal:`

`\if_mode_vertical:`

`\if_mode_inner:`

```
872 \cs_set_eq:NwN \if_mode_math: \tex_ifmmode:D
873 \cs_set_eq:NwN \if_mode_horizontal: \tex_ifhmode:D
874 \cs_set_eq:NwN \if_mode_vertical: \tex_ifvmode:D
875 \cs_set_eq:NwN \if_mode_inner: \tex_ifinner:D
```

(End definition for `\if_mode_math:.` This function is documented on page 9.)

`\if_cs_exist:N`

`\if_cs_exist:w`

```
876 \cs_set_eq:NwN \if_cs_exist:N \etex_ifdefined:D
877 \cs_set_eq:NwN \if_cs_exist:w \etex_ifcsname:D
```

(End definition for `\if_cs_exist:N`. This function is documented on page 9.)

`\exp_after:wN`

The three `\exp_` functions are used in the `l3expan` module where they are described.

`\exp_not:N`

`\exp_not:n`

```
878 \cs_set_eq:NwN \exp_after:wN \tex_expandafter:D
879 \cs_set_eq:NwN \exp_not:N \tex_noexpand:D
880 \cs_set_eq:NwN \exp_not:n \etex_unexpanded:D
```

(End definition for `\exp_after:wN`. This function is documented on page 31.)

`\iow_shipout_x:Nn`

`\token_to_meaning:N`

`\token_to_str:N`

`\token_to_str:c`

`\cs:w`

`\cs_end:`

`\cs_meaning:N`

`\cs_meaning:c`

`\cs_show:N`

`\cs_show:c`

```
881 \cs_set_eq:NwN \iow_shipout_x:Nn \tex_write:D
882 \cs_set_eq:NwN \token_to_meaning:N \tex_meaning:D
883 \cs_set_eq:NwN \token_to_str:N \tex_string:D
884 \cs_set_eq:NwN \cs:w \tex_csname:D
885 \cs_set_eq:NwN \cs_end: \tex_endcsname:D
886 \cs_set_eq:NwN \cs_meaning:N \tex_meaning:D
887 \tex_def:D \cs_meaning:c {\exp_args:Nc\cs_meaning:N}
888 \cs_set_eq:NwN \cs_show:N \tex_show:D
889 \tex_def:D \cs_show:c {\exp_args:Nc\cs_show:N}
890 \tex_def:D \token_to_str:c {\exp_args:Nc\token_to_str:N}
```

(End definition for `\iow_shipout_x:Nn`. This function is documented on page 12.)

`\scan_stop:`

`\group_begin:`

`\group_end:`

The next three are basic functions for which there also exist versions that are safe inside alignments. These safe versions are defined in the `l3prg` module.

```
891 \cs_set_eq:NwN \scan_stop: \tex_relax:D
892 \cs_set_eq:NwN \group_begin: \tex_begingroup:D
893 \cs_set_eq:NwN \group_end: \tex_endgroup:D
```

(End definition for `\scan_stop:`. This function is documented on page 25.)

`\group_execute_after:N`

```
894 \cs_set_eq:NwN \group_execute_after:N \tex_aftergroup:D
```

(End definition for `\group_execute_after:N`. This function is documented on page 25.)

`\pref_global:D`
`\pref_long:D`
`\pref_protected:D`

```
895 \cs_set_eq:NwN \pref_global:D \tex_global:D
896 \cs_set_eq:NwN \pref_long:D \tex_long:D
897 \cs_set_eq:NwN \pref_protected:D \etex_protected:D
```

(End definition for `\pref_global:D`. This function is documented on page 23.)

95.2 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

`\cs_set_nopar:Npn`
`\cs_set_nopar:Npx`
`\cs_set:Npn`
`\cs_set:Npx`
`\cs_set_protected_nopar:Npn`
`\cs_set_protected_nopar:Npx`
`\cs_set_protected:Npn`
`\cs_set_protected:Npx`

All assignment functions in L^AT_EX₃ should be naturally robust; after all, the T_EX primitives for assignments are and it can be a cause of problems if others aren't.

```
898 \cs_set_eq:NwN \cs_set_nopar:Npn \tex_def:D
899 \cs_set_eq:NwN \cs_set_nopar:Npx \tex_edef:D
900 \pref_protected:D \cs_set_nopar:Npn \cs_set:Npn {
901   \pref_long:D \cs_set_nopar:Npn
902 }
903 \pref_protected:D \cs_set_nopar:Npn \cs_set:Npx {
904   \pref_long:D \cs_set_nopar:Npx
905 }
906 \pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn {
907   \pref_protected:D \cs_set_nopar:Npn
908 }
909 \pref_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx {
910   \pref_protected:D \cs_set_nopar:Npx
911 }
912 \cs_set_protected_nopar:Npn \cs_set_protected:Npn {
913   \pref_protected:D \pref_long:D \cs_set_nopar:Npn
914 }
915 \cs_set_protected_nopar:Npn \cs_set_protected:Npx {
916   \pref_protected:D \pref_long:D \cs_set_nopar:Npx
917 }
```

(End definition for `\cs_set_nopar:Npn`. This function is documented on page 20.)

`\cs_gset_nopar:Npn`
`\cs_gset_nopar:Npx`
`\cs_gset:Npn`
`\cs_gset:Npx`
`\cs_gset_protected_nopar:Npn`
`\cs_gset_protected_nopar:Npx`
`\cs_gset_protected:Npn`
`\cs_gset_protected:Npx`

Global versions of the above functions.

```
918 \cs_set_eq:NwN \cs_gset_nopar:Npn \tex_gdef:D
```

```

919 \cs_set_eq:NwN \cs_gset_nopar:Npx \tex_xdef:D
920 \cs_set_protected_nopar:Npn \cs_gset:Npn {
921   \pref_long:D \cs_gset_nopar:Npn
922 }
923 \cs_set_protected_nopar:Npn \cs_gset:Npx {
924   \pref_long:D \cs_gset_nopar:Npx
925 }
926 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn {
927   \pref_protected:D \cs_gset_nopar:Npn
928 }
929 \cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx {
930   \pref_protected:D \cs_gset_nopar:Npx
931 }
932 \cs_set_protected_nopar:Npn \cs_gset_protected:Npn {
933   \pref_protected:D \pref_long:D \cs_gset_nopar:Npn
934 }
935 \cs_set_protected_nopar:Npn \cs_gset_protected:Npx {
936   \pref_protected:D \pref_long:D \cs_gset_nopar:Npx
937 }

```

(End definition for `\cs_gset_nopar:Npn`. This function is documented on page 20.)

95.3 Selecting tokens

`\use:c` This macro grabs its argument and returns a csname from it.

```

938 \cs_set:Npn \use:c #1 { \cs:w#1\cs_end: }

```

(End definition for `\use:c`. This function is documented on page 13.)

`\use:x` Fully expands its argument and passes it to the input stream. Uses `\cs_tmp:` as a scratch register but does not affect it.

```

939 \cs_set_protected:Npn \use:x #1 {
940   \group_begin:
941     \cs_set:Npx \cs_tmp: {#1}
942     \exp_after:wN
943   \group_end:
944   \cs_tmp:
945 }

```

(End definition for `\use:x`. This function is documented on page 13.)

`\use:n` These macro grabs its arguments and returns it back to the input (with outer braces removed). `\use:n` is defined earlier for bootstrapping.

```

946 \cs_set:Npn \use:n #1 {#1}
947 \cs_set:Npn \use:nn #1#2 {#1#2}
948 \cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
949 \cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}

```


(End definition for `\use:n`. This function is documented on page 12.)

`\use_i:nn` `\use_ii:nn` These macros are needed to provide functions with true and false cases, as introduced by Michael some time ago. By using `\exp_after:wN \use_i:nn \else:` constructions it is possible to write code where the true or false case is able to access the following tokens from the input stream, which is not possible if the `\c_true_bool` syntax is used.

```
950 \cs_set:Npn \use_i:nn #1#2 {#1}
951 \cs_set:Npn \use_ii:nn #1#2 {#2}
```

(End definition for `\use_i:nn`. This function is documented on page 13.)

`\use_i:nnn` `\use_ii:nnn` `\use_iii:nnn` `\use_i:nnnn` `\use_ii:nnnn` `\use_iii:nnnn` `\use_iv:nnnn` `\use_i_ii:nnn` We also need something for picking up arguments from a longer list.

```
952 \cs_set:Npn \use_i:nnn #1#2#3{#1}
953 \cs_set:Npn \use_ii:nnn #1#2#3{#2}
954 \cs_set:Npn \use_iii:nnn #1#2#3{#3}
955 \cs_set:Npn \use_i:nnnn #1#2#3#4{#1}
956 \cs_set:Npn \use_ii:nnnn #1#2#3#4{#2}
957 \cs_set:Npn \use_iii:nnnn #1#2#3#4{#3}
958 \cs_set:Npn \use_iv:nnnn #1#2#3#4{#4}
959 \cs_set:Npn \use_i_ii:nnn #1#2#3{#1#2}
```

(End definition for `\use_i:nnn`. This function is documented on page 14.)

`\use_none_delimit_by_q_nil:w` `\use_none_delimit_by_q_stop:w` `\use_none_delimit_by_q_recursion_stop:w` Functions that gobble everything until they see either `\q_nil` or `\q_stop` resp.

```
960 \cs_set:Npn \use_none_delimit_by_q_nil:w #1\q_nil{}
961 \cs_set:Npn \use_none_delimit_by_q_stop:w #1\q_stop{}
962 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop {}
```

(End definition for `\use_none_delimit_by_q_nil:w`. This function is documented on page 14.)

`\use_i_delimit_by_q_nil:nw` `\use_i_delimit_by_q_stop:nw` `\use_i_delimit_by_q_recursion_stop:nw` Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```
963 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2\q_nil{#1}
964 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2\q_stop{#1}
965 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #1#2 \q_recursion_stop {#1}
```

(End definition for `\use_i_delimit_by_q_nil:nw`. This function is documented on page 14.)

`\use_i_after_fi:nw` `\use_i_after_else:nw` `\use_i_after_or:nw` `\use_i_after_orelse:nw` Returns the first argument after ending the conditional.

```
966 \cs_set:Npn \use_i_after_fi:nw #1\fi:{\fi: #1}
967 \cs_set:Npn \use_i_after_else:nw #1\else:#2\fi:{\fi: #1}
968 \cs_set:Npn \use_i_after_or:nw #1\or: #2\fi: {\fi:#1}
969 \cs_set:Npn \use_i_after_orelse:nw #1 #2#3\fi: {\fi:#1}
```

(End definition for `\use_i_after_fi:nw`. This function is documented on page 15.)

95.4 Gobbling tokens from input

```

\use_none:n
\use_none:nn
\use_none:nnn
\use_none:nnnn
\use_none:nnnnn
\use_none:nnnnnn
\use_none:nnnnnnn
\use_none:nnnnnnnn

```

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of `n`'s following the `:` in the name. Although defining `\use_none:nnn` and above as separate calls of `\use_none:n` and `\use_none:nn` is slightly faster, this is very non-intuitive to the programmer who will assume that expanding such a function once will take care of gobbling all the tokens in one go.

```

970 \cs_set:Npn \use_none:n #1{}
971 \cs_set:Npn \use_none:nn #1#2{}
972 \cs_set:Npn \use_none:nnn #1#2#3{}
973 \cs_set:Npn \use_none:nnnn #1#2#3#4{}
974 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5{}
975 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6{}
976 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7{}
977 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8{}
978 \cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9{}

```

(End definition for `\use_none:n`. This function is documented on page 13.)

95.5 Expansion control from `l3expan`

```

\exp_args:Nc

```

Moved here for now as it is going to be used right away.

```

979 \cs_set:Npn \exp_args:Nc #1#2{\exp_after:wN#1\cs:w#2\cs_end:}

```

(End definition for `\exp_args:Nc`. This function is documented on page 29.)

95.6 Conditional processing and definitions

Underneath any predicate function (`_p`) or other conditional forms (TF, etc.) is a built-in logic saying that it after all of the testing and processing must return the *state* this leaves `TeX` in. Therefore, a simple user interface could be something like

```

\if_meaning:w #1#2 \prg_return_true: \else:
\if_meaning:w #1#3 \prg_return_true: \else:
\prg_return_false:
\fi: \fi:

```

Usually, a `TeX` programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the `TeX` programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

`\prg_return_true:` The idea here is that `\tex_roman numeral:D` will expand fully any `\tex_else:D` and the
`\prg_return_false:` `\tex_fi:D` that are waiting to be discarded, before reaching the `\c_zero` which will leave
the expansion null. The code can then leave either the first or second argument in the
input stream. This means that all of the branching code has to contain at least two
tokens: see how the logical tests are actually implemented to see this.

```

980 \cs_set_nopar:Npn \prg_return_true:
981   { \exp_after:wN \use_i:nn \tex_roman numeral:D }
982 \cs_set_nopar:Npn \prg_return_false:
983   { \exp_after:wN \use_ii:nn \tex_roman numeral:D }

```

An extended state space could be implemented by including a more elaborate function in
place of `\use_i:nn/\use_ii:nn`. Provided two arguments are absorbed then the code
will work.

(End definition for `\prg_return_true:`. This function is documented on page 33.)

`\prg_set_conditional:Npnn` The user functions for the types using parameter text from the programmer. Call aux
`\prg_new_conditional:Npnn` function to grab parameters, split the base function into name and signature and then
`\prg_set_protected_conditional:Npnn` use, e.g., `\cs_set:Npn` to define it with.
`\prg_new_protected_conditional:Npnn`

```

984 \cs_set_protected:Npn \prg_set_conditional:Npnn #1{
985   \prg_get_parm_aux:nw{
986     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
987     \cs_set:Npn {parm}
988   }
989 }
990 \cs_set_protected:Npn \prg_new_conditional:Npnn #1{
991   \prg_get_parm_aux:nw{
992     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
993     \cs_new:Npn {parm}
994   }
995 }
996 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn #1{
997   \prg_get_parm_aux:nw{
998     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
999     \cs_set_protected:Npn {parm}
1000  }
1001 }
1002 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn #1{
1003   \prg_get_parm_aux:nw{
1004     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1005     \cs_new_protected:Npn {parm}
1006   }
1007 }

```

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 34.)

`\prg_set_conditional:Nnn` The user functions for the types automatically inserting the correct parameter text based
`\prg_new_conditional:Nnn`
`\prg_set_protected_conditional:Nnn`
`\prg_new_protected_conditional:Nnn`

on the signature. Call aux function after calculating number of arguments, split the base function into name and signature and then use, e.g., `\cs_set:Npn` to define it with.

```

1008 \cs_set_protected:Npn \prg_set_conditional:Nnn #1{
1009   \exp_args:Nnf \prg_get_count_aux:nn{
1010     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1011     \cs_set:Npn {count}
1012   }\cs_get_arg_count_from_signature:N #1}
1013 }
1014 \cs_set_protected:Npn \prg_new_conditional:Nnn #1{
1015   \exp_args:Nnf \prg_get_count_aux:nn{
1016     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1017     \cs_new:Npn {count}
1018   }\cs_get_arg_count_from_signature:N #1}
1019 }
1020
1021 \cs_set_protected:Npn \prg_set_protected_conditional:Nnn #1{
1022   \exp_args:Nnf \prg_get_count_aux:nn{
1023     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1024     \cs_set_protected:Npn {count}
1025   }\cs_get_arg_count_from_signature:N #1}
1026 }
1027
1028 \cs_set_protected:Npn \prg_new_protected_conditional:Nnn #1{
1029   \exp_args:Nnf \prg_get_count_aux:nn{
1030     \cs_split_function:NN #1 \prg_generate_conditional_aux:nnNNnnnn
1031     \cs_new_protected:Npn {count}
1032   }\cs_get_arg_count_from_signature:N #1}
1033 }

```

(End definition for `\prg_set_conditional:Nnn` and others. These functions are documented on page 34.)

`\prg_set_eq_conditional:NNn`
`\prg_new_eq_conditional:NNn`

The obvious setting-equal functions.

```

1034 \cs_set_protected:Npn \prg_set_eq_conditional:NNn #1#2#3 {
1035   \prg_set_eq_conditional_aux:NNNn \cs_set_eq:cc #1#2 {#3}
1036 }
1037 \cs_set_protected:Npn \prg_new_eq_conditional:NNn #1#2#3 {
1038   \prg_set_eq_conditional_aux:NNNn \cs_new_eq:cc #1#2 {#3}
1039 }

```

(End definition for `\prg_set_eq_conditional:NNn` and `\prg_new_eq_conditional:NNn`. These functions are documented on page 34.)

`\prg_get_parm_aux:nw`
`\prg_get_count_aux:nn`

For the `Npnn` type we must grab the parameter text before continuing. We make this a very generic function that takes one argument before reading everything up to a left brace. Something similar for the `Nnn` type.

```

1040 \cs_set:Npn \prg_get_count_aux:nn #1#2 {#1{#2}}
1041 \cs_set:Npn \prg_get_parm_aux:nw #1#2#{#1{#2}}

```

(End definition for `\prg_get_parm_aux:nw` and `\prg_get_count_aux:nn`.)

`\prg_generate_conditional_parm_aux:nnNNnnnn`
`\prg_generate_conditional_parm_aux:nw`

The workhorse here is going through a list of desired forms, i.e., p, TF, T and F. The first three arguments come from splitting up the base form of the conditional, which gives the name, signature and a boolean to signal whether or not there was a colon in the name. For the time being, we do not use this piece of information but could well throw an error. The fourth argument is how to define this function, the fifth is the text `parm` or `count` for which version to use to define the functions, the sixth is the parameters to use (possibly empty) or number of arguments, the seventh is the list of forms to define, the eighth is the replacement text which we will augment when defining the forms.

```

1042 \cs_set:Npn \prg_generate_conditional_aux:nnNNnnnn #1#2#3#4#5#6#7#8{
1043   \prg_generate_conditional_aux:nnw{#5}{
1044     #4{#1}{#2}{#6}{#8}
1045   }#7,?, \q_recursion_stop
1046 }
```

Looping through the list of desired forms. First is the text `parm` or `count`, second is five arguments packed together and third is the form. Use text and form to call the correct type.

```

1047 \cs_set:Npn \prg_generate_conditional_aux:nnw #1#2#3,{
1048   \if:w ?#3
1049     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1050   \fi:
1051   \use:c{prg_generate_#3_form_#1:Nnnnn} #2
1052   \prg_generate_conditional_aux:nnw{#1}{#2}
1053 }
```

(End definition for `\prg_generate_conditional_parm_aux:nnNNnnnn` and `\prg_generate_conditional_parm_aux:nw`.)

`\prg_generate_p_form_parm:Nnnnn`
`\prg_generate_TF_form_parm:Nnnnn`
`\prg_generate_T_form_parm:Nnnnn`
`\prg_generate_F_form_parm:Nnnnn`

How to generate the various forms. The `parm` types here takes the following arguments: 1: how to define (an N-type), 2: name, 3: signature, 4: parameter text (or empty), 5: replacement. Remember that the logic-returning functions expect two arguments to be present after `\c_zero`: notice the construction of the different variants relies on this, and that the TF variant will be slightly faster than the T version.

```

1054 \cs_set_protected:Npn \prg_generate_p_form_parm:Nnnnn #1#2#3#4#5
1055 {
1056   \exp_args:Nc #1 { #2 _p: #3 } #4
1057   {
1058     #5 \c_zero
1059     \c_true_bool \c_false_bool
1060   }
1061 }
1062 \cs_set_protected:Npn \prg_generate_T_form_parm:Nnnnn #1#2#3#4#5
1063 {
1064   \exp_args:Nc #1 { #2 : #3 T } #4
1065   {
```

```

1066         #5 \c_zero
1067         \use:n \use_none:n
1068     }
1069 }
1070 \cs_set_protected:Npn \prg_generate_F_form_parm:Nnnnn #1#2#3#4#5
1071 {
1072     \exp_args:Nc #1 { #2 : #3 F } #4
1073     {
1074         #5 \c_zero
1075         { }
1076     }
1077 }
1078 \cs_set_protected:Npn \prg_generate_TF_form_parm:Nnnnn #1#2#3#4#5
1079 {
1080     \exp_args:Nc #1 { #2 : #3 TF } #4
1081     { #5 \c_zero }
1082 }

```

(End definition for \prg_generate_p_form_parm:Nnnnn and others.)

\prg_generate_p_form_count:Nnnnn The **count** form is similar, but of course requires a number rather than a primitive argument specification.

```

\prg_generate_TF_form_count:Nnnnn
\prg_generate_T_form_count:Nnnnn
\prg_generate_F_form_count:Nnnnn
1083 \cs_set_protected:Npn \prg_generate_p_form_count:Nnnnn #1#2#3#4#5
1084 {
1085     \cs_generate_from_arg_count:cNnn { #2 _p: #3 } #1 {#4}
1086     {
1087         #5 \c_zero
1088         \c_true_bool \c_false_bool
1089     }
1090 }
1091 \cs_set_protected:Npn \prg_generate_T_form_count:Nnnnn #1#2#3#4#5
1092 {
1093     \cs_generate_from_arg_count:cNnn { #2 : #3 T } #1 {#4}
1094     {
1095         #5 \c_zero
1096         \use:n \use_none:n
1097     }
1098 }
1099 \cs_set_protected:Npn \prg_generate_F_form_count:Nnnnn #1#2#3#4#5
1100 {
1101     \cs_generate_from_arg_count:cNnn { #2 : #3 F } #1 {#4}
1102     {
1103         #5 \c_zero
1104         { }
1105     }
1106 }
1107 \cs_set_protected:Npn \prg_generate_TF_form_count:Nnnnn #1#2#3#4#5
1108 {
1109     \cs_generate_from_arg_count:cNnn { #2 : #3 TF } #1 {#4}

```

```

1110      { #5 \c_zero }
1111    }

```

(End definition for \prg_generate_p_form_count:Nnnnn and others.)

```

\prg_set_eq_conditional_aux:NNNn
\prg_set_eq_conditional_aux:NNNw

```

```

1112 \cs_set:Npn \prg_set_eq_conditional_aux:NNNn #1#2#3#4 {
1113   \prg_set_eq_conditional_aux:NNNw #1#2#3#4,?,\q_recursion_stop
1114 }

```

Manual clist loop over argument #4.

```

1115 \cs_set:Npn \prg_set_eq_conditional_aux:NNNw #1#2#3#4, {
1116   \if:w ? #4 \scan_stop:
1117     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1118   \fi:
1119   #1 {
1120     \exp_args:Nnc \cs_split_function:NN #2 {prg_conditional_form_#4:nnn}
1121   }{
1122     \exp_args:Nnc \cs_split_function:NN #3 {prg_conditional_form_#4:nnn}
1123   }
1124   \prg_set_eq_conditional_aux:NNNw #1{#2}{#3}
1125 }

1126 \cs_set:Npn \prg_conditional_form_p:nnn #1#2#3 {#1_p:#2}
1127 \cs_set:Npn \prg_conditional_form_TF:nnn #1#2#3 {#1:#2TF}
1128 \cs_set:Npn \prg_conditional_form_T:nnn #1#2#3 {#1:#2T}
1129 \cs_set:Npn \prg_conditional_form_F:nnn #1#2#3 {#1:#2F}

```

(End definition for \prg_set_eq_conditional_aux:NNNn and \prg_set_eq_conditional_aux:NNNw.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using \if:w but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

\c_true_bool Here are the canonical boolean values.
\c_false_bool

```

1130 \tex_chardef:D \c_true_bool = 1~
1131 \tex_chardef:D \c_false_bool = 0~

```

(End definition for \c_true_bool. This function is documented on page 11.)

95.7 Dissecting a control sequence

\cs_to_str:N This converts a control sequence into the character string of its name, removing the
\cs_to_str_aux:w leading escape character. This turns out to be a non-trivial matter as there a different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of `\tex_escapechar:D` is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N__` yields the escape character itself and a space. The escape sequence will terminate the expansion started by `\tex_romannumeral:D`, which is a negative number and so will not gobble the escape character even if it's a number. The `\tex_if:D` test will then be `false`, and the naïve approach of gobbling the first character of the `\token_to_str:N` version of the control sequence will work, even if the first character is a space. The second case is that the escape character is itself a space. In this case, the escape character space is consumed terminating the first `\tex_romannumeral:D`, and `\cs_to_str_aux:w` is expanded. This inserts a space, making the `\tex_if:D` test `true`. The second `\tex_romannumeral:D` will then execute the `\token_to_str:N`, with the escape-character space being consumed by the `\tex_romannumeral:D`, and thus leaving the control sequence name in the input stream. The final case is where the escape character is not printable. The flow here starts with the `\token_to_str:N__` giving just a space, which terminates the first `\tex_romannumeral:D` but leaves no token for the `\tex_if:D` test. This means that the `\tex_romannumeral:D` is executed before the test is finished. The result is that the `\tex_fi:D`, expanded before the `\tex_if:D` is finished, becomes `\tex_relax:D \tex_fi:D`, and the `\tex_relax:D` is then used in the `\tex_if:D` test. In this case, `\token_to_str:N` is therefore used with no gobbling at all, which is exactly what is needed in this case.

```

1132 \cs_set:Npn \cs_to_str:N
1133 {
1134   \tex_if:D \tex_romannumeral:D - '0 \token_to_str:N \ %
1135     \cs_to_str_aux:w
1136   \tex_fi:D
1137   \exp_after:wN \use_none:n \token_to_str:N
1138 }
1139 \cs_set_nopar:Npn \cs_to_str_aux:w #1 \use_none:n
1140 { ~ \tex_romannumeral:D - '0 \tex_fi:D }
```

(End definition for `\cs_to_str:N`. This function is documented on page 24.)

`\cs_split_function:NN` This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean `<true>` or `<false>` is returned with `<true>` for when there is a colon in the function and `<false>`

```

\cs_split_function_aux:w
\cs_split_function_auxii:w
```


if there is not. Lastly, the second argument of `\cs_split_function:NN` is supposed to be a function taking three variables, one for name, one for signature, and one for the boolean. For example, `\cs_split_function:NN\foo_bar:cnx\use_i:nnn` as input becomes `\use_i:nnn {foo_bar}{cnx}\c_true_bool`.

Can't use a literal `:` because it has the wrong catcode here, so it's transformed from `@` with `\tex_lowercase:D`.

```

1141 \group_begin:
1142   \tex_lccode:D  '\@ = '\: \scan_stop:
1143   \tex_catcode:D '\@ = 12~
1144   \tex_lowercase:D {
1145   \group_end:

```

First ensure that we actually get a properly evaluated str as we don't know how many expansions `\cs_to_str:N` requires. Insert extra colon to catch the error cases.

```

1146 \cs_set:Npn \cs_split_function:NN #1#2{
1147   \exp_after:wN \cs_split_function_aux:w
1148   \tex_romannumeral:D -'\q \cs_to_str:N #1 @a \q_stop #2
1149 }

```

If no colon in the name, #2 is a with catcode 11 and #3 is empty. If colon in the name, then either #2 is a colon or the first letter of the signature. The letters here have catcode 12. If a colon was given we need to a) split off the colon and quark at the end and b) ensure we return the name, signature and boolean true. We can't use `\quark_if_no_value:NTF` yet but this is very safe anyway as all tokens have catcode 12.

```

1150 \cs_set:Npn \cs_split_function_aux:w #1@#2#3\q_stop#4{
1151   \if_meaning:w a#2
1152     \exp_after:wN \use_i:nn
1153   \else:
1154     \exp_after:wN \use_ii:nn
1155   \fi:
1156   {#4{#1}{}}\c_false_bool}
1157   {\cs_split_function_auxii:w#2#3\q_stop #4{#1}}
1158 }
1159 \cs_set:Npn \cs_split_function_auxii:w #1@a\q_stop#2#3{
1160   #2{#3}{#1}\c_true_bool
1161 }

```

End of lowercase

```

1162 }

```

(End definition for `\cs_split_function:NN`. This function is documented on page 24.)

`\cs_get_function_name:N` Now returning the name is trivial: just discard the last two arguments. Similar for `\cs_get_function_signature:N` signature.

```

1163 \cs_set:Npn \cs_get_function_name:N #1 {

```

```

1164 \cs_split_function:NN #1\use_i:nnn
1165 }
1166 \cs_set:Npn \cs_get_function_signature:N #1 {
1167   \cs_split_function:NN #1\use_ii:nnn
1168 }

```

(End definition for `\cs_get_function_name:N` and `\cs_get_function_signature:N`. These functions are documented on page 24.)

95.8 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive `\tex_relax:D` token. A control sequence is said to be *free* (to be defined) if it does not already exist and also meets the requirement that it does not contain a D signature. The reasoning behind this is that most of the time, a check for a free control sequence is when we wish to make a new control sequence and we do not want to let the user define a new “do not use” control sequence.

`\cs_if_exist_p:N` Two versions for checking existence. For the N form we firstly check for `\tex_relax:D`
`\cs_if_exist_p:c` and then if it is in the hash table. There is no problem when inputting something like
`\cs_if_exist:NTF` `\else:` or `\fi:` as T_EX will only ever skip input in case the token tested against is
`\cs_if_exist:cTF` `\tex_relax:D`.

```

1169 \prg_set_conditional:Npnn \cs_if_exist:N #1 {p,TF,T,F}{
1170   \if_meaning:w #1\tex_relax:D
1171   \prg_return_false:
1172   \else:
1173     \if_cs_exist:N #1
1174     \prg_return_true:
1175     \else:
1176       \prg_return_false:
1177   \fi:
1178 \fi:
1179 }

```

For the c form we firstly check if it is in the hash table and then for `\tex_relax:D` so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```

1180 \prg_set_conditional:Npnn \cs_if_exist:c #1 {p,TF,T,F}{
1181   \if_cs_exist:w #1 \cs_end:
1182   \exp_after:wN \use_i:nn
1183   \else:
1184     \exp_after:wN \use_ii:nn
1185   \fi:
1186 {

```

```

1187 \exp_after:wN \if_meaning:w \cs:w #1\cs_end: \tex_relax:D
1188 \prg_return_false:
1189 \else:
1190 \prg_return_true:
1191 \fi:
1192 }
1193 \prg_return_false:
1194 }

```

(End definition for `\cs_if_exist_p:N` and `\cs_if_exist_p:c`. These functions are documented on page 10.)

`\cs_if_do_not_use_p:N`
`\cs_if_do_not_use_aux:nnN`

```

1195 \cs_set:Npn \cs_if_do_not_use_p:N #1{
1196 \cs_split_function:NN #1 \cs_if_do_not_use_aux:nnN
1197 }
1198 \cs_set:Npn \cs_if_do_not_use_aux:nnN #1#2#3{
1199 \str_if_eq_p:nn { D } {#2}
1200 }

```

(End definition for `\cs_if_do_not_use_p:N`. This function is documented on page 10.)

`\cs_if_free_p:N`
`\cs_if_free_p:c`
`\cs_if_free:NTF`
`\cs_if_free:cTF`

The simple implementation is one using the boolean expression parser: If it exists or is do not use, then return false.

```

\prg_set_conditional:Npnn \cs_if_free:N #1{p,TF,T,F}{
  \bool_if:nTF {\cs_if_exist_p:N #1 || \cs_if_do_not_use_p:N #1}
    {\prg_return_false:}\prg_return_true:}
}

```

However, this functionality may not be available this early on. We do something similar: The numerical values of true and false is one and zero respectively, which we can use. The problem again here is that the token we are checking may in fact be something that can disturb the scanner, so we have to be careful. We would like to do minimal evaluation so we ensure this.

```

1201 \prg_set_conditional:Npnn \cs_if_free:N #1{p,TF,T,F}{
1202 \tex_ifnum:D \cs_if_exist_p:N #1 =\c_zero
1203 \exp_after:wN \use_i:nn
1204 \else:
1205 \exp_after:wN \use_ii:nn
1206 \fi:
1207 {
1208 \tex_ifnum:D \cs_if_do_not_use_p:N #1 =\c_zero
1209 \prg_return_true:
1210 \else:
1211 \prg_return_false:

```

```

1212     \fi:
1213   }
1214   \prg_return_false:
1215 }
1216 \cs_set_nopar:Npn \cs_if_free_p:c{\exp_args:Nc\cs_if_free_p:N}
1217 \cs_set_nopar:Npn \cs_if_free:cTF{\exp_args:Nc\cs_if_free:NTF}
1218 \cs_set_nopar:Npn \cs_if_free:cT{\exp_args:Nc\cs_if_free:NT}
1219 \cs_set_nopar:Npn \cs_if_free:cF{\exp_args:Nc\cs_if_free:NF}

```

(End definition for `\cs_if_free_p:N` and `\cs_if_free_p:c`. These functions are documented on page 10.)

95.9 Defining and checking (new) functions

`\c_minus_one` We need the constants `\c_minus_one` and `\c_sixteen` now for writing information to the log and the terminal and `\c_zero` which is used by some functions in the `l3alloc` module.
`\c_zero`
`\c_sixteen` The rest are defined in the `l3int` module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the `l3int` module is required but it can't be used until the allocation has been set up properly! The actual allocation mechanism is in `l3alloc` and as \TeX wants to reserve count registers 0–9, the first available one is 10 so we use that for `\c_minus_one`.
`\c_six`
`\c_seven`
`\c_twelve`

```

1220 <!*initex>
1221 \cs_set_eq:NwN \c_minus_one\m@ne
1222 <!/initex>
1223 <!*package>
1224 \tex_countdef:D \c_minus_one = 10 ~
1225 \c_minus_one = -1 ~
1226 <!/package>
1227 \tex_chardef:D \c_sixteen = 16~
1228 \tex_chardef:D \c_zero = 0~
1229 \tex_chardef:D \c_six = 6~
1230 \tex_chardef:D \c_seven = 7~
1231 \tex_chardef:D \c_twelve = 12~

```

(End definition for `\c_minus_one`, `\c_zero`, and `\c_sixteen`. These functions are documented on page 66.)

`\c_max_register_int` This is here as this particular integer is needed both in package mode and to bootstrap `l3alloc`

```

1232 \tex_mathchardef:D \c_max_register_int = 32767 \tex_relax:D

```

(End definition for `\c_max_register_int`. This function is documented on page 66.)

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called `\..._new`. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The definitions here are only temporary, they will be redefined later on.

`\iow_log:x` We define a routine to write only to the log file. And a similar one for writing to both
`\iow_term:x` the log file and the terminal.

```

1233 \cs_set_protected_nopar:Npn \iow_log:x {
1234   \tex_immediate:D \iow_shipout_x:Nn \c_minus_one
1235 }
1236 \cs_set_protected_nopar:Npn \iow_term:x {
1237   \tex_immediate:D \iow_shipout_x:Nn \c_sixteen
1238 }
```

(End definition for `\iow_log:x`. This function is documented on page 120.)

`\msg_kernel_bug:x` This will show internal errors.

```

1239 \cs_set_protected_nopar:Npn \msg_kernel_bug:x #1 {
1240   \iow_term:x { This~is~a~LaTeX~bug:~check~coding! }
1241   \tex_errmessage:D {#1}
1242 }
```

(End definition for `\msg_kernel_bug:x`. This function is documented on page 129.)

`\cs_record_meaning:N` This macro will be used later on for tracing purposes. But we need some more modules to define it, so we just give some dummy definition here.

```

1243 <*trace>
1244 \cs_set:Npn \cs_record_meaning:N #1{}
1245 </trace>
```

(End definition for `\cs_record_meaning:N`. This function is documented on page 16.)

`\chk_if_free_cs:N` This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:NN` etc. to make sure
`\chk_if_free_cs:c` that the argument sequence is not already in use. If it is, an error is signalled. It checks if `<csname>` is undefined or `\scan_stop:.` Otherwise an error message is issued. We have to make sure we don't put the argument into the conditional processing since it may be an `\if...` type function!

```

1246 \cs_set_protected_nopar:Npn \chk_if_free_cs:N #1{
1247   \cs_if_free:NF #1
1248   {
1249     \msg_kernel_bug:x {Command~name~'\token_to_str:N #1'~already~defined!~
1250                       Current~meaning:~\ \c_space_tl \c_space_tl \token_to_meaning:N #1
1251                       }
1252   }
1253 <*trace>
1254   \cs_record_meaning:N#1
1255   % \iow_term:x{Defining~\token_to_str:N #1~on~%}
```

```

1256 \iow_log:x{Defining~\token_to_str:N #1~on~
1257             line~\tex_the:D \tex_inputlineno:D}
1258 \trace>
1259 }
1260 \cs_set_protected_nopar:Npn \chk_if_free_cs:c {
1261   \exp_args:Nc \chk_if_free_cs:N
1262 }
1263 <*package>
1264 \tex_ifodd:D \@l@expl@log@functions@bool \else
1265   \cs_set_protected_nopar:Npn \chk_if_free_cs:N #1 {
1266     \cs_if_free:NF #1
1267     {
1268       \msg_kernel_bug:x
1269       {
1270         Command~name~'\token_to_str:N #1'~already~defined!~
1271         Current~meaning: \ \c_space_tl \c_space_tl \token_to_meaning:N #1
1272       }
1273     }
1274   }
1275 \fi
1276 </package>

```

(End definition for `\chk_if_free_cs:N`. This function is documented on page 11.)

`\chk_if_exist_cs:N` This function issues a warning message when the control sequence in its argument does not exist.

```

1277 \cs_set_protected_nopar:Npn \chk_if_exist_cs:N #1 {
1278   \cs_if_exist:NF #1
1279   {
1280     \msg_kernel_bug:x {Command~ '\token_to_str:N #1'~
1281                       not~ yet~ defined!}
1282   }
1283 }
1284 \cs_set_protected_nopar:Npn \chk_if_exist_cs:c {
1285   \exp_args:Nc \chk_if_exist_cs:N
1286 }

```

(End definition for `\chk_if_exist_cs:N`. This function is documented on page 11.)

`\str_if_eq_p:nn` Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore make life a bit clearer. The `nn` and `xx` versions are created directly as this is most efficient.

`\str_if_eq:nnTF`
`\str_if_eq_p:xx`
`\str_if_eq:xxTF`

```

1287 \prg_set_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF } {
1288   \tex_ifnum:D \pdf_strcmp:D
1289   { \etex_unexpanded:D {#1} } { \etex_unexpanded:D {#2} }
1290   = \c_zero
1291   \prg_return_true: \else: \prg_return_false: \fi:
1292 }

```

(End definition for `\str_if_eq_p:nn`. This function is documented on page 11.)

```
1297 \prg_set_conditional:Npnn \cs_if_eq_name:NN #1#2{p}{
1298   \str_if_eq_p:nn {#1} {#2}
1299 }
```

(End definition for `\cs_if_eq_name_p:NN`. This function is documented on page 10.)

95.10 More new definitions

`\cs_new_nopar:Npn` Global versions of the above functions.

(End definition for `\cs_new_nopar:Npn`. This function is documented on page 17.)

<code>\cs_set_nopar:cpn</code>	Like <code>\cs_set_nopar:Npn</code> and <code>\cs_new_nopar:Npn</code> , except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the <code>c</code> stands for <code>csname</code> argument, see the expansion module). Global versions are also provided.
<code>\cs_set_nopar:cpx</code>	
<code>\cs_gset_nopar:cpn</code>	
<code>\cs_gset_nopar:cpx</code>	

`\cs_new_nopar:cpn` $\langle string \rangle \langle rep-text \rangle$ will turn $\langle string \rangle$ into a csname and then assign $\langle rep-text \rangle$ to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

1315 \cs_set:Npn \cs_tmp:w #1#2{
1316   \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 }
1317 }

```

```

1318 \cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1319 \cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1320 \cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1321 \cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1322 \cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1323 \cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End definition for `\cs_set_nopar:cpn`. This function is documented on page 17.)

`\cs_set:cpn` Variants of the `\cs_set:Npn` versions which make a csname out of the first arguments.
`\cs_set:cpx` We may also do this globally.

```

\cs_gset:cpn
\cs_gset:cpx
\cs_new:cpn
\cs_new:cpx
1324 \cs_tmp:w \cs_set:cpn \cs_set:Npn
1325 \cs_tmp:w \cs_set:cpx \cs_set:Npx
1326 \cs_tmp:w \cs_gset:cpn \cs_gset:Npn
1327 \cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1328 \cs_tmp:w \cs_new:cpn \cs_new:Npn
1329 \cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End definition for `\cs_set:cpn`. This function is documented on page 17.)

`\cs_set_protected_nopar:cpn` Variants of the `\cs_set_protected_nopar:Npn` versions which make a csname out of
`\cs_set_protected_nopar:cpx` the first arguments. We may also do this globally.
`\cs_gset_protected_nopar:cpn`
`\cs_gset_protected_nopar:cpx`
`\cs_new_protected_nopar:cpn`
`\cs_new_protected_nopar:cpx`

```

1330 \cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
1331 \cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
1332 \cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1333 \cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1334 \cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
1335 \cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx

```

(End definition for `\cs_set_protected_nopar:cpn`. This function is documented on page 18.)

`\cs_set_protected:cpn` Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first
`\cs_set_protected:cpx` arguments. We may also do this globally.
`\cs_gset_protected:cpn`
`\cs_gset_protected:cpx`
`\cs_new_protected:cpn`
`\cs_new_protected:cpx`

```

1336 \cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
1337 \cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
1338 \cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
1339 \cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1340 \cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
1341 \cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

(End definition for `\cs_set_protected:cpn`. This function is documented on page 17.)

BACKWARDS COMPATIBILITY:

```

1342 \cs_set_eq:NwN \cs_gnew_nopar:Npn \cs_new_nopar:Npn
1343 \cs_set_eq:NwN \cs_gnew:Npn \cs_new:Npn
1344 \cs_set_eq:NwN \cs_gnew_protected_nopar:Npn \cs_new_protected_nopar:Npn
1345 \cs_set_eq:NwN \cs_gnew_protected:Npn \cs_new_protected:Npn

```



```

1346 \cs_set_eq:NwN          \cs_gnew_nopar:Npx          \cs_new_nopar:Npx
1347 \cs_set_eq:NwN          \cs_gnew:Npx              \cs_new:Npx
1348 \cs_set_eq:NwN \cs_gnew_protected_nopar:Npx \cs_new_protected_nopar:Npx
1349 \cs_set_eq:NwN          \cs_gnew_protected:Npx      \cs_new_protected:Npx
1350 \cs_set_eq:NwN          \cs_gnew_nopar:cpn          \cs_new_nopar:cpn
1351 \cs_set_eq:NwN          \cs_gnew:cpn                \cs_new:cpn
1352 \cs_set_eq:NwN \cs_gnew_protected_nopar:cpn \cs_new_protected_nopar:cpn
1353 \cs_set_eq:NwN          \cs_gnew_protected:cpn      \cs_new_protected:cpn
1354 \cs_set_eq:NwN          \cs_gnew_nopar:cpx          \cs_new_nopar:cpx
1355 \cs_set_eq:NwN          \cs_gnew:cpx                \cs_new:cpx
1356 \cs_set_eq:NwN \cs_gnew_protected_nopar:cpx \cs_new_protected_nopar:cpx
1357 \cs_set_eq:NwN          \cs_gnew_protected:cpx      \cs_new_protected:cpx

```

\use_0_parameter: For using parameters, i.e., when you need to define a function to process three parameters.
\use_1_parameter: See `xparse` for an application.

```

\use_2_parameter:
\use_3_parameter:
\use_4_parameter:
\use_5_parameter:
\use_6_parameter:
\use_7_parameter:
\use_8_parameter:
\use_9_parameter:
1358 \cs_set_nopar:cpn{use_0_parameter:}{ }
1359 \cs_set_nopar:cpn{use_1_parameter:}{{##1}}
1360 \cs_set_nopar:cpn{use_2_parameter:}{{##1}{##2}}
1361 \cs_set_nopar:cpn{use_3_parameter:}{{##1}{##2}{##3}}
1362 \cs_set_nopar:cpn{use_4_parameter:}{{##1}{##2}{##3}{##4}}
1363 \cs_set_nopar:cpn{use_5_parameter:}{{##1}{##2}{##3}{##4}{##5}}
1364 \cs_set_nopar:cpn{use_6_parameter:}{{##1}{##2}{##3}{##4}{##5}{##6}}
1365 \cs_set_nopar:cpn{use_7_parameter:}{{##1}{##2}{##3}{##4}{##5}{##6}{##7}}
1366 \cs_set_nopar:cpn{use_8_parameter:}{
1367   {##1}{##2}{##3}{##4}{##5}{##6}{##7}{##8}}
1368 \cs_set_nopar:cpn{use_9_parameter:}{
1369   {##1}{##2}{##3}{##4}{##5}{##6}{##7}{##8}{##9}}

```

(End definition for `\use_0_parameter:.`)

95.11 Copying definitions

`\cs_set_eq:NN` These macros allow us to copy the definition of a control sequence to another control sequence.

`\cs_set_eq:cN`
`\cs_set_eq:Nc`
`\cs_set_eq:cc` The = sign allows us to define funny char tokens like = itself or `_` with this function. For the definition of `\c_space_chartok{~}` to work we need the `~` after the =.

`\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it long in order to throw an ‘already defined’ error rather than ‘runaway argument’.

The `c` variants are not protected in order for their arguments to be constructed in the correct context.

```

1370 \cs_set_protected:Npn \cs_set_eq:NN #1 { \cs_set_eq:NwN #1=~ }
1371 \cs_set_protected_nopar:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
1372 \cs_set_protected_nopar:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
1373 \cs_set_protected_nopar:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }

```

(End definition for `\cs_set_eq:NN`. This function is documented on page 23.)

```

\cs_new_eq:NN
\cs_new_eq:cN
\cs_new_eq:Nc
\cs_new_eq:cc
1374 \cs_new_protected:Npn \cs_new_eq:NN #1 {
1375   \chk_if_free_cs:N #1
1376   \pref_global:D \cs_set_eq:NN #1
1377 }
1378 \cs_new_protected_nopar:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1379 \cs_new_protected_nopar:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
1380 \cs_new_protected_nopar:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }

```

(End definition for `\cs_new_eq:NN`. This function is documented on page 22.)

```

\cs_gset_eq:NN
\cs_gset_eq:cN
\cs_gset_eq:Nc
\cs_gset_eq:cc
1381 \cs_new_protected:Npn \cs_gset_eq:NN { \pref_global:D \cs_set_eq:NN }
1382 \cs_new_protected_nopar:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
1383 \cs_new_protected_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1384 \cs_new_protected_nopar:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }

```

(End definition for `\cs_gset_eq:NN`. This function is documented on page 23.)

BACKWARDS COMPATIBILITY

```

1385 \cs_set_eq:NN \cs_gnew_eq:NN \cs_new_eq:NN
1386 \cs_set_eq:NN \cs_gnew_eq:cN \cs_new_eq:cN
1387 \cs_set_eq:NN \cs_gnew_eq:Nc \cs_new_eq:Nc
1388 \cs_set_eq:NN \cs_gnew_eq:cc \cs_new_eq:cc

```

95.12 Undefining functions

`\cs_undefine:N` The following function is used to free the main memory from the definition of some function that isn't in use any longer.

```

\cs_undefine:c
\cs_gundefine:N
\cs_gundefine:c
1389 \cs_new_protected_nopar:Npn \cs_undefine:N #1 {
1390   \cs_set_eq:NN #1 \c_undefined:D
1391 }
1392 \cs_new_protected_nopar:Npn \cs_undefine:c #1 {
1393   \cs_set_eq:cN {#1} \c_undefined:D
1394 }
1395 \cs_new_protected_nopar:Npn \cs_gundefine:N #1 {
1396   \cs_gset_eq:NN #1 \c_undefined:D
1397 }
1398 \cs_new_protected_nopar:Npn \cs_gundefine:c #1 {
1399   \cs_gset_eq:cN {#1} \c_undefined:D
1400 }

```

(End definition for `\cs_undefine:N` and `\cs_undefine:c`. These functions are documented on page 22.)

95.13 Diagnostic wrapper functions

```

\kernel_register_show:N
\kernel_register_show:c
1401 \cs_new_nopar:Npn \kernel_register_show:N #1 {
1402   \cs_if_exist:NTF #1
1403   {
1404     \tex_showthe:D #1
1405   }
1406   {
1407     \msg_kernel_bug:x {Register~ '\token_to_str:N #1'~ is~ not~ defined.}
1408   }
1409 }
1410 \cs_new_nopar:Npn \kernel_register_show:c { \exp_args:Nc \int_show:N }

```

(End definition for `\kernel_register_show:N` and `\kernel_register_show:c`. These functions are documented on page ??.)

95.14 Engine specific definitions

```

\c_xetex_is_engine_bool
\c_luatex_is_engine_bool
\etex_if_engine:TF
\luatex_if_engine:TF

```

In some cases it will be useful to know which engine we're running. Don't provide a `_p` predicate because the `_bool` is used for the same thing.

```

1411 \cs_if_exist:NTF \xetex_version:D
1412 { \cs_new_eq:NN \c_xetex_is_engine_bool \c_true_bool }
1413 { \cs_new_eq:NN \c_xetex_is_engine_bool \c_false_bool }
1414 \prg_new_conditional:Npnn \etex_if_engine: {TF,T,F} {
1415   \if_bool:N \c_xetex_is_engine_bool
1416   \prg_return_true: \else: \prg_return_false: \fi:
1417 }

1418 \cs_if_exist:NTF \luatex_directlua:D
1419 { \cs_new_eq:NN \c_luatex_is_engine_bool \c_true_bool }
1420 { \cs_new_eq:NN \c_luatex_is_engine_bool \c_false_bool }
1421 \prg_set_conditional:Npnn \etex_if_engine: {TF,T,F}{
1422   \if_bool:N \c_xetex_is_engine_bool \prg_return_true:
1423   \else: \prg_return_false: \fi:
1424 }
1425 \prg_set_conditional:Npnn \luatex_if_engine: {TF,T,F}{
1426   \if_bool:N \c_luatex_is_engine_bool \prg_return_true:
1427   \else: \prg_return_false: \fi:
1428 }

```

(End definition for `\c_xetex_is_engine_bool` and `\c_luatex_is_engine_bool`. These functions are documented on page 25.)

95.15 Scratch functions

```
\prg_do_nothing:
```

I don't think this function belongs here, but one place is as good as any other. I want to use this function when I want to express 'no operation'. It is for example used in

templates where depending on the users settings we have to either select an function that does something, or one that does nothing.

```
1429 \cs_new_nopar:Npn \prg_do_nothing: {}
```

(End definition for `\prg_do_nothing:`. This function is documented on page 15.)

95.16 Defining functions from a given number of arguments

```
\cs_get_arg_count_from_signature:N
\cs_get_arg_count_from_signature_aux:nnN
\cs_get_arg_count_from_signature_auxii:w
```

Counting the number of tokens in the signature, i.e., the number of arguments the function should take. If there is no signature, we return that there is -1 arguments to signal an error. Otherwise we insert the string 9876543210 after the signature. If the signature is empty, the number we want is 0 so we remove the first nine tokens and return the tenth. Similarly, if the signature is `nnn` we want to remove the nine tokens `nnn987654` and return 3. Therefore, we simply remove the first nine tokens and then return the tenth.

```
1430 \cs_set:Npn \cs_get_arg_count_from_signature:N #1{
1431   \cs_split_function:NN #1 \cs_get_arg_count_from_signature_aux:nnN
1432 }
1433 \cs_set:Npn \cs_get_arg_count_from_signature_aux:nnN #1#2#3{
1434   \if_predicate:w #3 % \bool_if:NTF here
1435     \exp_after:wN \use_i:nn
1436   \else:
1437     \exp_after:wN\use_ii:nn
1438   \fi:
1439   {
1440     \exp_after:wN \cs_get_arg_count_from_signature_auxii:w
1441     \use_none:nnnnnnnn #2 9876543210\q_stop
1442   }
1443   {-1}
1444 }
1445 \cs_set:Npn \cs_get_arg_count_from_signature_auxii:w #1#2\q_stop{#1}
```

A variant form we need right away.

```
1446 \cs_set_nopar:Npn \cs_get_arg_count_from_signature:c {
1447   \exp_args:Nc \cs_get_arg_count_from_signature:N
1448 }
```

(End definition for `\cs_get_arg_count_from_signature:N`. This function is documented on page 24.)

```
\cs_generate_from_arg_count:NNnn
\cs_generate_from_arg_count_error_msg:Nn
```

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since \TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```

1449 \cs_set:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4{
1450   \tex_ifcase:D \etex_numexpr:D #3\tex_relax:D
1451     \use_i_after_orelse:nw{#2#1}
1452   \or:
1453     \use_i_after_orelse:nw{#2#1 ##1}
1454   \or:
1455     \use_i_after_orelse:nw{#2#1 ##1##2}
1456   \or:
1457     \use_i_after_orelse:nw{#2#1 ##1##2##3}
1458   \or:
1459     \use_i_after_orelse:nw{#2#1 ##1##2##3##4}
1460   \or:
1461     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5}
1462   \or:
1463     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6}
1464   \or:
1465     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7}
1466   \or:
1467     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7##8}
1468   \or:
1469     \use_i_after_orelse:nw{#2#1 ##1##2##3##4##5##6##7##8##9}
1470   \else:
1471     \use_i_after_fi:nw{
1472       \cs_generate_from_arg_count_error_msg:Nn#1{#3}
1473       \use_none:n % to remove replacement text
1474     }
1475   \fi:
1476   {#4}
1477 }

```

A variant form we need right away.

```

1478 \cs_set_nopar:Npn \cs_generate_from_arg_count:cNnn {
1479   \exp_args:Nc \cs_generate_from_arg_count:NNnn
1480 }

```

The error message. Elsewhere we use the value of -1 to signal a missing colon in a function, so provide a hint for help on this.

```

1481 \cs_set:Npn \cs_generate_from_arg_count_error_msg:Nn #1#2 {
1482   \msg_kernel_bug:x {
1483     You're~ trying~ to~ define~ the~ command~ '\token_to_str:N #1'~
1484     with~ \use:n{\tex_the:D\etex_numexpr:D #2\tex_relax:D} ~
1485     arguments~ but~ I~ only~ allow~ 0-9~arguments.~Perhaps~you~
1486     forgot~to~use~a~colon~in~the~function~name?~
1487     I~ can~ probably~ not~ help~ you~ here
1488   }
1489 }

```

(End definition for `\cs_generate_from_arg_count:NNnn`.)

95.17 Using the signature to define functions

We can now combine some of the tools we have to provide a simple interface for defining functions. We define some simpler functions with user interface `\cs_set:Nn \foo_bar:nn {#1,#2}`, i.e., the number of arguments is read from the signature.

We want to define `\cs_set:Nn` as

```

\cs_set:Nn
\cs_set:Nx
\cs_set_nopar:Nn
\cs_set_nopar:Nx
\cs_set_protected:Nn
\cs_set_protected:Nx
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Nx
\cs_gset:Nn
\cs_gset:Nx
\cs_gset_nopar:Nn
\cs_gset_nopar:Nx
\cs_gset_protected:Nn
\cs_gset_protected:Nx
\cs_gset_protected_nopar:Nn
\cs_gset_protected_nopar:Nx

```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```

1490 \cs_set:Npn \cs_tmp:w #1#2#3{
1491   \cs_set_protected:cpx {cs_#1:#2}##1##2{
1492     \exp_not:N \cs_generate_from_arg_count:NNnn ##1
1493     \exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:
1494     {\exp_not:N\cs_get_arg_count_from_signature:N ##1}{##2}
1495   }
1496 }

```

Then we define the 32 variants beginning with N.

```

1497 \cs_tmp:w {set}{Nn}{Npn}
1498 \cs_tmp:w {set}{Nx}{Npx}
1499 \cs_tmp:w {set_nopar}{Nn}{Npn}
1500 \cs_tmp:w {set_nopar}{Nx}{Npx}
1501 \cs_tmp:w {set_protected}{Nn}{Npn}
1502 \cs_tmp:w {set_protected}{Nx}{Npx}
1503 \cs_tmp:w {set_protected_nopar}{Nn}{Npn}
1504 \cs_tmp:w {set_protected_nopar}{Nx}{Npx}
1505 \cs_tmp:w {gset}{Nn}{Npn}
1506 \cs_tmp:w {gset}{Nx}{Npx}
1507 \cs_tmp:w {gset_nopar}{Nn}{Npn}
1508 \cs_tmp:w {gset_nopar}{Nx}{Npx}
1509 \cs_tmp:w {gset_protected}{Nn}{Npn}
1510 \cs_tmp:w {gset_protected}{Nx}{Npx}
1511 \cs_tmp:w {gset_protected_nopar}{Nn}{Npn}
1512 \cs_tmp:w {gset_protected_nopar}{Nx}{Npx}

```

(End definition for `\cs_set:Nn`. This function is documented on page 22.)

```

\cs_new:Nn
\cs_new:Nx
\cs_new_nopar:Nn
\cs_new_nopar:Nx
\cs_new_protected:Nn
\cs_new_protected:Nx
\cs_new_protected_nopar:Nn
\cs_new_protected_nopar:Nx

```

```

1513 \cs_tmp:w {new}{Nn}{Npn}
1514 \cs_tmp:w {new}{Nx}{Npx}
1515 \cs_tmp:w {new_nopar}{Nn}{Npn}
1516 \cs_tmp:w {new_nopar}{Nx}{Npx}
1517 \cs_tmp:w {new_protected}{Nn}{Npn}
1518 \cs_tmp:w {new_protected}{Nx}{Npx}
1519 \cs_tmp:w {new_protected_nopar}{Nn}{Npn}
1520 \cs_tmp:w {new_protected_nopar}{Nx}{Npx}

```

(End definition for `\cs_new:Nn`. This function is documented on page 19.)

Then something similar for the c variants.

```

\cs_set_protected:Npn \cs_set:cn #1#2{
  \cs_generate_from_arg_count:cNnn {#1}\cs_set:Npn
    {\cs_get_arg_count_from_signature:c {#1}}{#2}
}

```

```

1521 \cs_set:Npn \cs_tmp:w #1#2#3{
1522   \cs_set_protected:cpx {cs_#1:#2}##1##2{
1523     \exp_not:N\cs_generate_from_arg_count:cNnn {##1}
1524     \exp_after:wN \exp_not:N \cs:w cs_#1:#3 \cs_end:
1525     {\exp_not:N\cs_get_arg_count_from_signature:c {##1}}{##2}
1526   }
1527 }

```

```

\cs_set:cn
\cs_set:cx
\cs_set_nopar:cn
\cs_set_nopar:cx
\cs_set_protected:cn
\cs_set_protected:cx
\cs_set_protected_nopar:cn
\cs_set_protected_nopar:cx
\cs_gset:cn
\cs_gset:cx
\cs_gset_nopar:cn
\cs_gset_nopar:cx
\cs_gset_protected:cn
\cs_gset_protected:cx
\cs_gset_protected_nopar:cn
\cs_gset_protected_nopar:cx

```

The 32 c variants.

```

1528 \cs_tmp:w {set}{cn}{Npn}
1529 \cs_tmp:w {set}{cx}{Npx}
1530 \cs_tmp:w {set_nopar}{cn}{Npn}
1531 \cs_tmp:w {set_nopar}{cx}{Npx}
1532 \cs_tmp:w {set_protected}{cn}{Npn}
1533 \cs_tmp:w {set_protected}{cx}{Npx}
1534 \cs_tmp:w {set_protected_nopar}{cn}{Npn}
1535 \cs_tmp:w {set_protected_nopar}{cx}{Npx}
1536 \cs_tmp:w {gset}{cn}{Npn}
1537 \cs_tmp:w {gset}{cx}{Npx}
1538 \cs_tmp:w {gset_nopar}{cn}{Npn}
1539 \cs_tmp:w {gset_nopar}{cx}{Npx}
1540 \cs_tmp:w {gset_protected}{cn}{Npn}
1541 \cs_tmp:w {gset_protected}{cx}{Npx}
1542 \cs_tmp:w {gset_protected_nopar}{cn}{Npn}
1543 \cs_tmp:w {gset_protected_nopar}{cx}{Npx}

```

(End definition for `\cs_set:cn`. This function is documented on page 22.)

```

\cs_new:cn
\cs_new:cx
\cs_new_nopar:cn
\cs_new_nopar:cx
\cs_new_protected:cn
\cs_new_protected:cx
\cs_new_protected_nopar:cn
\cs_new_protected_nopar:cx
1544 \cs_tmp:w {new}{cn}{Npn}
1545 \cs_tmp:w {new}{cx}{Npx}
1546 \cs_tmp:w {new_nopar}{cn}{Npn}
1547 \cs_tmp:w {new_nopar}{cx}{Npx}
1548 \cs_tmp:w {new_protected}{cn}{Npn}
1549 \cs_tmp:w {new_protected}{cx}{Npx}
1550 \cs_tmp:w {new_protected_nopar}{cn}{Npn}
1551 \cs_tmp:w {new_protected_nopar}{cx}{Npx}

```

(End definition for `\cs_new:cn`. This function is documented on page 19.)

```

\cs_if_eq_p:NN
\cs_if_eq_p:cN
\cs_if_eq_p:Nc
\cs_if_eq_p:cc
\cs_if_eq:NNTF
\cs_if_eq:cNTF
\cs_if_eq:NcTF
\cs_if_eq:ccTF
1552 \prg_set_conditional:Npnn \cs_if_eq:NN #1#2{p,TF,T,F}{
1553   \if_meaning:w #1#2
1554   \prg_return_true: \else: \prg_return_false: \fi:
1555 }
1556 \cs_new_nopar:Npn \cs_if_eq_p:cN {\exp_args:Nc \cs_if_eq_p:NN}
1557 \cs_new_nopar:Npn \cs_if_eq:cNTF {\exp_args:Nc \cs_if_eq:NNTF}
1558 \cs_new_nopar:Npn \cs_if_eq:cNT {\exp_args:Nc \cs_if_eq:NNT}
1559 \cs_new_nopar:Npn \cs_if_eq:cNF {\exp_args:Nc \cs_if_eq:NNF}
1560 \cs_new_nopar:Npn \cs_if_eq_p:Nc {\exp_args:NNc \cs_if_eq_p:NN}
1561 \cs_new_nopar:Npn \cs_if_eq:NcTF {\exp_args:NNc \cs_if_eq:NNTF}
1562 \cs_new_nopar:Npn \cs_if_eq:NcT {\exp_args:NNc \cs_if_eq:NNT}
1563 \cs_new_nopar:Npn \cs_if_eq:NcF {\exp_args:NNc \cs_if_eq:NNF}
1564 \cs_new_nopar:Npn \cs_if_eq_p:cc {\exp_args:Ncc \cs_if_eq_p:NN}
1565 \cs_new_nopar:Npn \cs_if_eq:ccTF {\exp_args:Ncc \cs_if_eq:NNTF}
1566 \cs_new_nopar:Npn \cs_if_eq:ccT {\exp_args:Ncc \cs_if_eq:NNT}
1567 \cs_new_nopar:Npn \cs_if_eq:ccF {\exp_args:Ncc \cs_if_eq:NNF}

```

(End definition for `\cs_if_eq_p:NN` and others. These functions are documented on page 10.)

```

1568 </initex | package>

```

96 l3expan implementation

96.1 Internal functions and variables

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

This will expand $\langle token_2 \rangle$ once before processing $\langle token_1 \rangle$. This is similar to `\exp_args:No` except that no braces are put around the result of expanding $\langle token_2 \rangle$.

T_EXhackers note: This is the primitive `\expandafter` which was renamed to fit into the naming conventions of L^AT_EX3.

`\l_exp_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.

<code>\exp_eval_register:N *</code> <code>\exp_eval_register:c *</code>	<code>\exp_eval_register:N <register></code>
--	--

These functions evaluates a register as part of a `V` or `v` expansion (respectively). A register might exist as one of two things: A parameter-less non-long, non-protected macro or a built-in `TEX` register such as `\count`.

<code>\exp_eval_error_msg:w</code>	<code>\exp_eval_error_msg:w <register></code>
------------------------------------	---

Used to generate an error message if a variable called as part of a `v` or `V` expansion is defined as `\scan_stop:`. This typically indicates that an incorrect cs name has been used.

<code>\n::</code> <code>\N::</code> <code>\c::</code> <code>\o::</code> <code>\f::</code> <code>\x::</code> <code>\v::</code> <code>\V::</code> <code>\::</code>	<code>\cs_set_nopar:Npn \exp_args:Ncof {\\::c\\::o\\::f\\::}</code>
--	---

Internal forms for the base expansion types.

96.2 Module code

We start by ensuring that the required packages are loaded.

```

1569 <*package>
1570 \ProvidesExplPackage
1571   {\filename}{\filedate}{\fileversion}{\filedescription}
1572 \package_check_loaded_expl:
1573 </package>
1574 <*initex | package>

```

`\exp_after:wN` These are defined in `l3basics`.

`\exp_not:N`
`\exp_not:n`

```

1575 <*bootstrap>
1576 \cs_set_eq:NwN \exp_after:wN \tex_expandafter:D

```

```

1577 \cs_set_eq:NwN \exp_not:N \tex_noexpand:D
1578 \cs_set_eq:NwN \exp_not:n \etex_unexpanded:D
1579 </bootstrap>

```

(End definition for `\exp_after:wN`. This function is documented on page 31.)

96.3 General expansion

In this section a general mechanism for defining functions to handle argument handling is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the L^AT_EX3 names for `\cs_set_nopar:Npx` at some point, and so is never going to be expandable.⁸)

The definition of expansion functions with this technique happens in section 96.5. In section 96.4 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

`\l_exp_tl` We need a scratch token list variable. We don't use `tl` methods so that `l3expan` can be loaded earlier.

```

1580 \cs_new_nopar:Npn \l_exp_tl {}

```

(End definition for `\l_exp_tl`. This function is documented on page ??.)

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are long as this turned out to be desirable since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\::<Z>` always has signature `#1\:::#2#3` where `#1` holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, `#2` is the carried over result of the previous expansion steps and `#3` is the argument about to be processed.

`\exp_arg_next:nnn` `#1` is the result of an expansion step, `#2` is the remaining argument manipulations and `#3` is the current result of the expansion chain. This auxiliary function moves `#1` back after `#3` in the input stream and checks if any expansion is left to be done by calling `#2`. In by far the most cases we will require to add a set of braces to the result of an argument manipulation so it is more effective to do it directly here. Actually, so far only the `c` of the final argument manipulation variants does not require a set of braces.

```

1581 \cs_new:Npn\exp_arg_next:nnn#1#2#3{
1582   #2\:::{#3{#1}}
1583 }
1584 \cs_new:Npn\exp_arg_next_nobrace:nnn#1#2#3{
1585   #2\:::{#3#1}
1586 }

```

⁸However, some primitives have certain characteristics that means that their arguments undergo an `x` type expansion but the primitive is in fact still expandable. We shall make it very clear when such a function is expandable.

(End definition for `\exp_arg_next:nnn`.)

\::: The end marker is just another name for the identity function.

```
1587 \cs_new:Npn\:::#1{#1}
```

(End definition for `\:::`. This function is documented on page 210.)

\::n This function is used to skip an argument that doesn't need to be expanded.

```
1588 \cs_new:Npn\::n#1\:::#2#3{
1589   #1\:::{#2{#3}}
1590 }
```

(End definition for `\::n`. This function is documented on page ??.)

\::N This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
1591 \cs_new:Npn\::N#1\:::#2#3{
1592   #1\:::{#2#3}
1593 }
```

(End definition for `\::N`. This function is documented on page ??.)

\::c This function is used to skip an argument that is turned into a control sequence without expansion.

```
1594 \cs_new:Npn\::c#1\:::#2#3{
1595   \exp_after:wN\exp_arg_next_nobraces:nnn\cs:w #3\cs_end:{#1}{#2}
1596 }
```

(End definition for `\::c`. This function is documented on page ??.)

\::o This function is used to expand an argument once.

```
1597 \cs_new:Npn\::o#1\:::#2#3{
1598   \exp_after:wN\exp_arg_next_nnn\exp_after:wN{#3}{#1}{#2}
1599 }
```

(End definition for `\::o`. This function is documented on page ??.)

\::f This function is used to expand a token list until the first unexpandable token is found.
\exp_stop_f: The underlying `\tex_romannumeral:D` -'0 expands everything in its way to find something terminating the number and thereby expands the function in front of it. This scanning procedure is terminated once the expansion hits something non-expandable or a space. We introduce `\exp_stop_f:` to mark such an end of expansion marker; in case the scanner hits a number, this number also terminates the scanning and is left untouched. In the example shown earlier the scanning was stopped once $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ had fully expanded

`\cs_set_eq:Nc \aaa {b \l_tmpa_tl b}` into `\cs_set_eq:NwN \aaa = \blurb` which then turned out to contain the non-expandable token `\cs_set_eq:NwN`. Since the expansion of `\tex_romannumeral:D -'0` is $\langle null \rangle$, we wind up with a fully expanded list, only TeX has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```

1600 \cs_new:Npn \::f#1\::#2#3{
1601   \exp_after:wN \exp_arg_next:nnn
1602   \exp_after:wN {\tex_romannumeral:D -'0 #3}
1603   {#1}{#2}
1604 }
1605 \cs_new_nopar:Npn \exp_stop_f: {~}

```

(End definition for `\::f`. This function is documented on page 32.)

\::x This function is used to expand an argument fully. We could use the new expandable primitive `\expanded` here, but we don't want to create incompatibilities between engines.

```

1606 \cs_new_protected:Npn \::x #1 \::: #2#3 {
1607   \cs_set_nopar:Npx \l_exp_tl {{#3}}
1608   \exp_after:wN \exp_arg_next:nnn \l_exp_tl {#1}{#2}
1609 }

```

(End definition for `\::x`. This function is documented on page ??.)

\::v These functions return the value of a register, i.e., one of `tl`, `num`, `int`, `skip`, `dim`
\::V and `muskip`. The `V` version expects a single token whereas `v` like `c` creates a `csname` from its argument given in braces and then evaluates it as if it was a `V`. The sequence `\tex_romannumeral:D -'0` sets off an `f` type expansion. The argument is returned in braces.

```

1610 \cs_new:Npn \::V#1\::: #2#3{
1611   \exp_after:wN \exp_arg_next:nnn
1612   \exp_after:wN {
1613     \tex_romannumeral:D -'0
1614     \exp_eval_register:N #3
1615   }
1616   {#1}{#2}
1617 }
1618 \cs_new:Npn \::v#1\::: #2#3{
1619   \exp_after:wN \exp_arg_next:nnn
1620   \exp_after:wN {
1621     \tex_romannumeral:D -'0
1622     \exp_eval_register:c {#3}
1623   }
1624   {#1}{#2}
1625 }

```

(End definition for `\::v`. This function is documented on page ??.)

`\exp_eval_register:N` This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in T_EX register such as `\count`. For the T_EX registers we have to utilize a `\tex_the:D` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\tex_the:D` and when not to. What we do here is try to find out whether the token will expand to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the register in question when it has been prefixed with `\exp_not:N` and the register itself. If it is a macro, the prefixed `\exp_not:N` will temporarily turn it into the primitive `\tex_relax:D`.

```

1626 \cs_set_nopar:Npn \exp_eval_register:N #1{
1627   \exp_after:wN \if_meaning:w \exp_not:N #1#1

```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\tex_relax:D`. In that case we throw an error. We could let T_EX do it for us but that would result in the rather obscure

! You can't use '`\relax`' after `\the`.

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```

1628   \if_meaning:w \tex_relax:D #1
1629   \exp_eval_error_msg:w
1630   \fi:

```

The next bit requires some explanation. The function must be initiated by the sequence `\tex_romannumeral:D -'0` and we want to terminate this expansion chain by inserting an `\exp_stop_f:` token. However, we have to expand the register `#1` before we do that. If it is a T_EX register, we need to execute the sequence `\exp_after:wN\exp_stop_f:\tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN\exp_stop_f: #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```

1631   \else:
1632     \exp_after:wN \use_i_ii:nnn
1633   \fi:
1634   \exp_after:wN \exp_stop_f: \tex_the:D #1
1635 }
1636 \cs_set_nopar:Npn \exp_eval_register:c #1{
1637   \exp_after:wN\exp_eval_register:N\cs:w #1\cs_end:
1638 }

```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```

! Undefined control sequence.
\exp_eval_error_msg:w ...erroneous variable used!

1.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}

```

```

1639 \group_begin:%
1640 \tex_catcode:D'\!=11\tex_relax:D%
1641 \tex_catcode:D'\ =11\tex_relax:D%
1642 \cs_gset:Npn\exp_eval_error_msg:w#1\tex_the:D#2{%
1643 \fi:\fi:\erroneous variable used!}%
1644 \group_end:%

```

(End definition for `\exp_eval_register:N` and `\exp_eval_register:c`. These functions are documented on page 210.)

96.4 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable and therefore allow to prefix them with `\pref_global:D` for example. This together with the fact that the ‘general’ concept above is slower means that we should convert whenever possible and perhaps remove all remaining occurrences by hand-encoding in the end.

`\exp_args:No`
`\exp_args:NNo`
`\exp_args:NNNo`

```

1645 \cs_new:Npn \exp_args:No #1#2{\exp_after:wN#1\exp_after:wN{#2}}
1646 \cs_new:Npn \exp_args:NNo #1#2#3{\exp_after:wN#1\exp_after:wN#2
1647 \exp_after:wN{#3}}
1648 \cs_new:Npn \exp_args:NNNo #1#2#3#4{\exp_after:wN#1\exp_after:wN#2
1649 \exp_after:wN#3\exp_after:wN{#4}}

```

(End definition for `\exp_args:No`. This function is documented on page 31.)

`\exp_args:Nc`
`\exp_args:cc`
`\exp_args:NNc`
`\exp_args:Ncc`
`\exp_args:Nccc`

Here are the functions that turn their argument into csnames but are expandable.

```

1650 \cs_set:Npn \exp_args:Nc #1#2{\exp_after:wN#1\cs:w#2\cs_end:}
1651 \cs_new:Npn \exp_args:cc #1#2{\cs:w #1\exp_after:wN\cs_end:\cs:w #2\cs_end:}
1652 \cs_new:Npn \exp_args:NNc #1#2#3{\exp_after:wN#1\exp_after:wN#2
1653 \cs:w#3\cs_end:}
1654 \cs_new:Npn \exp_args:Ncc #1#2#3{\exp_after:wN#1
1655 \cs:w#2\exp_after:wN\cs_end:\cs:w#3\cs_end:}
1656 \cs_new:Npn \exp_args:Nccc #1#2#3#4{\exp_after:wN#1
1657 \cs:w#2\exp_after:wN\cs_end:\cs:w#3\exp_after:wN
1658 \cs_end:\cs:w #4\cs_end:}

```

(End definition for `\exp_args:Nc` and others. These functions are documented on page 31.)

`\exp_args:Nco`

If we force that the third argument always has braces, we could implement this function with less tokens and only two arguments.

```

1659 \cs_new:Npn \exp_args:Nco #1#2#3{\exp_after:wN#1\cs:w#2\exp_after:wN
1660 \cs_end:\exp_after:wN{#3}}

```

(End definition for `\exp_args:Nco`. This function is documented on page 30.)

96.5 Definitions with the ‘general’ technique

`\exp_args:Nf`
`\exp_args:Nv`
`\exp_args:Nx`

```
1661 \cs_set_nopar:Npn \exp_args:Nf {\::f\:::}
1662 \cs_set_nopar:Npn \exp_args:Nv {\::v\:::}
1663 \cs_set_nopar:Npn \exp_args:Nv {\::V\:::}
1664 \cs_set_protected_nopar:Npn \exp_args:Nx {\::x\:::}
```

(End definition for `\exp_args:Nf` and others. These functions are documented on page 29.)

`\exp_args:NNV`
`\exp_args:NNv`
`\exp_args:NNf`
`\exp_args:NNx`
`\exp_args:NVV`
`\exp_args:Ncx`
`\exp_args:Nfo`
`\exp_args:Nff`
`\exp_args:Ncf`
`\exp_args:Nco`
`\exp_args:Nnf`
`\exp_args:Nno`
`\exp_args:NnV`
`\exp_args:Nnx`
`\exp_args:Noo`
`\exp_args:Noc`
`\exp_args:Nox`
`\exp_args:Nxo`
`\exp_args:Nxx`

Here are the actual function definitions, using the helper functions above.

```
1665 \cs_set_nopar:Npn \exp_args:NNf {\::N\::f\:::}
1666 \cs_set_nopar:Npn \exp_args:NNv {\::N\::v\:::}
1667 \cs_set_nopar:Npn \exp_args:NNV {\::N\::V\:::}
1668 \cs_set_protected_nopar:Npn \exp_args:NNx {\::N\::x\:::}
1669
1670 \cs_set_protected_nopar:Npn \exp_args:Ncx {\::c\::x\:::}
1671 \cs_set_nopar:Npn \exp_args:Nfo {\::f\::o\:::}
1672 \cs_set_nopar:Npn \exp_args:Nff {\::f\::f\:::}
1673 \cs_set_nopar:Npn \exp_args:Ncf {\::c\::f\:::}
1674 \cs_set_nopar:Npn \exp_args:Nnf {\::n\::f\:::}
1675 \cs_set_nopar:Npn \exp_args:Nno {\::n\::o\:::}
1676 \cs_set_nopar:Npn \exp_args:NnV {\::n\::V\:::}
1677 \cs_set_protected_nopar:Npn \exp_args:Nnx {\::n\::x\:::}
1678
1679 \cs_set_nopar:Npn \exp_args:Noc {\::o\::c\:::}
1680 \cs_set_nopar:Npn \exp_args:Noo {\::o\::o\:::}
1681 \cs_set_protected_nopar:Npn \exp_args:Nox {\::o\::x\:::}
1682
1683 \cs_set_nopar:Npn \exp_args:NVV {\::V\::V\:::}
1684
1685 \cs_set_protected_nopar:Npn \exp_args:Nxo {\::x\::o\:::}
1686 \cs_set_protected_nopar:Npn \exp_args:Nxx {\::x\::x\:::}
```

(End definition for `\exp_args:NNV` and others. These functions are documented on page 30.)

`\exp_args:Ncco`
`\exp_args:Nccx`
`\exp_args:Ncnx`
`\exp_args:NcNc`
`\exp_args:NcNo`
`\exp_args:NNno`
`\exp_args:NNNV`
`\exp_args:Nnno`
`\exp_args:Nnnx`
`\exp_args:Nnox`
`\exp_args:Nooo`
`\exp_args:Noox`
`\exp_args:Nnnc`
`\exp_args:NNnx`
`\exp_args:NNoo`
`\exp_args:NNox`

```
1687 \cs_set_nopar:Npn \exp_args:NNNV {\::N\::N\::V\:::}
1688
1689 \cs_set_nopar:Npn \exp_args:NNno {\::N\::n\::o\:::}
1690 \cs_set_protected_nopar:Npn \exp_args:NNnx {\::N\::n\::x\:::}
1691 \cs_set_nopar:Npn \exp_args:NNoo {\::N\::o\::o\:::}
1692 \cs_set_protected_nopar:Npn \exp_args:NNox {\::N\::o\::x\:::}
1693
1694 \cs_set_nopar:Npn \exp_args:Nnnx {\::n\::n\::c\:::}
1695 \cs_set_nopar:Npn \exp_args:Nnno {\::n\::n\::o\:::}
1696 \cs_set_protected_nopar:Npn \exp_args:Nnnx {\::n\::n\::x\:::}
```

```

1697 \cs_set_protected_nopar:Npn \exp_args:Nnox {\::n\::o\::x\::}
1698
1699 \cs_set_nopar:Npn \exp_args:NcNc {\::c\::N\::c\::}
1700 \cs_set_nopar:Npn \exp_args:NcNo {\::c\::N\::o\::}
1701 \cs_set_nopar:Npn \exp_args:Ncco {\::c\::c\::o\::}
1702 \cs_set_nopar:Npn \exp_args:Ncco {\::c\::c\::o\::}
1703 \cs_set_protected_nopar:Npn \exp_args:Nccx {\::c\::c\::x\::}
1704 \cs_set_protected_nopar:Npn \exp_args:Ncnx {\::c\::n\::x\::}
1705
1706 \cs_set_protected_nopar:Npn \exp_args:Noox {\::o\::o\::x\::}
1707 \cs_set_nopar:Npn \exp_args:Nooo {\::o\::o\::o\::}

```

(End definition for `\exp_args:Ncco` and others. These functions are documented on page 31.)

96.6 Preventing expansion

```

\exp_not:o
\exp_not:f
\exp_not:v
\exp_not:V
1708 \cs_new:Npn\exp_not:o#1{\exp_not:n\exp_after:wN{#1}}
1709 \cs_new:Npn\exp_not:f#1{
1710   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 #1}
1711 }
1712 \cs_new:Npn\exp_not:v#1{
1713   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 \exp_eval_register:c {#1}}
1714 }
1715 \cs_new:Npn\exp_not:V#1{
1716   \exp_not:n\exp_after:wN{\tex_romannumeral:D -'0 \exp_eval_register:N #1}
1717 }

```

(End definition for `\exp_not:o`. This function is documented on page 32.)

`\exp_not:c` A helper function.

```

1718 \cs_new:Npn\exp_not:c#1{\exp_after:wN\exp_not:N\cs:w#1\cs_end:}

```

(End definition for `\exp_not:c`. This function is documented on page 31.)

96.7 Defining function variants

```

\cs_generate_variant:Nn #1 : Base form of a function; e.g., \tl_set:Nn
\cs_generate_variant_aux:nnNn #2 : One or more variant argument specifiers; e.g., {Nx,c,cx}
\cs_generate_variant_aux:nnw
\cs_generate_variant_aux:N

```

Split up the original base function to grab its name and signature consisting of k letters. Then we wish to iterate through the list of variant argument specifiers, and for each one construct a new function name using the original base name, the variant signature consisting of l letters and the last $k - l$ letters of the base signature. For example, for a base function `\tl_set:Nn` which needs a `c` variant form, we want the new signature to be `cn`.


```

1719 \cs_new_protected:Npn \cs_generate_variant:Nn #1 {
1720   \chk_if_exist_cs:N #1
1721   \cs_split_function:NN #1 \cs_generate_variant_aux:nnNn
1722 }

```

We discard the boolean and then set off a loop through the desired variant forms.

```

1723 \cs_set:Npn \cs_generate_variant_aux:nnNn #1#2#3#4{
1724   \cs_generate_variant_aux:nnw {#1}{#2} #4,?,\q_recursion_stop
1725 }

```

Next is the real work to be done. We now have 1: base name, 2: base signature, 3: beginning of variant signature. To construct the new csname and the `\exp_args:Ncc` form, we need the variant signature. In our example, we wanted to discard the first two letters of the base signature because the variant form started with `cc`. This is the same as putting first `cc` in the signature and then `\use_none:nn` followed by the base signature `NNn`. We therefore call a small loop that outputs an `n` for each letter in the variant signature and use this to call the correct `\use_none:` variant. Firstly though, we check whether to terminate the loop.

```

1726 \cs_set:Npn \cs_generate_variant_aux:nnw #1 #2 #3, {
1727   \if:w ? #3
1728   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1729   \fi:

```

Then check if the variant form has already been defined.

```

1730   \cs_if_free:cTF {
1731     #1:#3\use:c {use_none:\cs_generate_variant_aux:N #3 ?}#2
1732   }
1733   {

```

If not, then define it and then additionally check if the `\exp_args:N` form needed is defined.

```

1734     \cs_generate_variant_aux:ccpx { #1 : #2 }
1735     {
1736       #1:#3 \use:c{use_none:\cs_generate_variant_aux:N #3 ?}#2
1737     }
1738     {
1739       \exp_not:c { exp_args:N #3} \exp_not:c {#1:#2}
1740     }
1741     \cs_generate_internal_variant:n {#3}
1742   }

```

Otherwise tell that it was already defined.

```

1743   {
1744     \iow_log:x{
1745       Variant~\token_to_str:c {
1746         #1:#3\use:c {use_none:\cs_generate_variant_aux:N #3 ?}#2

```

```

1747     }~already~defined;~ not~ changing~ it~on~line~
1748     \tex_the:D \tex_inputlineno:D
1749   }
1750 }

```

Recurse.

```

1751   \cs_generate_variant_aux:nnw{#1}{#2}
1752 }

```

The small loop for defining the required number of ns. Break when seeing a ?.

```

1753 \cs_set:Npn \cs_generate_variant_aux:N #1{
1754   \if:w ?#1 \exp_after:wN\use_none:nn \fi: n \cs_generate_variant_aux:N
1755 }

```

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 27.)

```

\cs_generate_variant_aux:Ncpx
\cs_generate_variant_aux:ccpx
\cs_generate_variant_aux:w

```

The idea here is to pick up protected parent functions, using the nature of the meaning string that they generate. The test here is almost the same as `\tl_if_empty:nTF`, but has to be hard-coded as that function is not yet available and because it has to match both long and short macros.

```

1756 \group_begin:
1757   \tex_lccode:D '\Z = '\d \scan_stop:
1758   \tex_lccode:D '\? ='\ \ \scan_stop:
1759   \tex_catcode:D '\P = 12 \scan_stop:
1760   \tex_catcode:D '\R = 12 \scan_stop:
1761   \tex_catcode:D '\O = 12 \scan_stop:
1762   \tex_catcode:D '\T = 12 \scan_stop:
1763   \tex_catcode:D '\E = 12 \scan_stop:
1764   \tex_catcode:D '\C = 12 \scan_stop:
1765   \tex_catcode:D '\Z = 12 \scan_stop:
1766 \tex_lowercase:D {
1767   \group_end:
1768   \cs_new_nopar:Npn \cs_generate_variant_aux:Ncpx #1
1769   {
1770     \exp_after:wN \cs_generate_variant_aux:w
1771     \tex_meaning:D #1 ? PROTECTEZ \q_stop
1772   }
1773   \cs_new_nopar:Npn \cs_generate_variant_aux:ccpx
1774   { \exp_args:Nc \cs_generate_variant_aux:Ncpx}
1775   \cs_new:Npn \cs_generate_variant_aux:w
1776   #1 ? PROTECTEZ #2 \q_stop
1777   {
1778     \exp_after:wN \tex_ifx:D \exp_after:wN
1779     \q_no_value \etex_detokenize:D {#1} \q_no_value
1780     \exp_after:wN \cs_new_protected_nopar:cpx
1781   \tex_else:D
1782     \exp_after:wN \cs_new_nopar:cpx

```

```

1783     \tex_fi:D
1784   }
1785 }

```

(End definition for `_cs_generate_variant_aux:Ncpx`.)

`\cs_generate_internal_variant:n` Test if `exp_args:N #1` is already defined and if not define it via the `\::` commands using the chars in `#1`

```

1786 \cs_new_protected:Npn \cs_generate_internal_variant:n #1 {
1787   \cs_if_free:cT { exp_args:N #1 }{

```

We use `new` to log the definition if we have to make one.

```

1788     \cs_new:cpx { exp_args:N #1 }
1789     { \cs_generate_internal_variant_aux:n #1 : }
1790   }
1791 }

```

(End definition for `\cs_generate_internal_variant:n`. This function is documented on page 27.)

`\cs_generate_internal_variant_aux:n` This command grabs char by char outputting `\::#1` (not expanded further) until we see a `..`. That colon is in fact also turned into `\:::` so that the required structure for `\exp_args...` commands is correctly terminated.

```

1792 \cs_new:Npn \cs_generate_internal_variant_aux:n #1 {
1793   \exp_not:c{::#1}
1794   \if_meaning:w #1 :
1795     \exp_after:wN \use_none:n
1796   \fi:
1797   \cs_generate_internal_variant_aux:n
1798 }

```

(End definition for `\cs_generate_internal_variant_aux:n`.)

96.8 Last-unbraced versions

`\exp_arg_last_unbraced:nn` There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

\::f_unbraced
\::o_unbraced
\::V_unbraced
\::v_unbraced

1799 \cs_new:Npn \exp_arg_last_unbraced:nn #1#2 { #2#1 }
1800 \cs_new:Npn \::f_unbraced \:::#1#2 {
1801   \exp_after:wN \exp_arg_last_unbraced:nn
1802   \exp_after:wN { \tex_romannumeral:D -'0 #2 } {#1}
1803 }
1804 \cs_new:Npn \::o_unbraced \:::#1#2 {
1805   \exp_after:wN \exp_arg_last_unbraced:nn \exp_after:wN {#2 }{#1}
1806 }
1807 \cs_new:Npn \::V_unbraced \:::#1#2 {

```

```

1808 \exp_after:wN \exp_arg_last_unbraced:nn
1809 \exp_after:wN { \tex_romannumeral:D -'0 \exp_eval_register:N #2 } {#1}
1810 }
1811 \cs_new:Npn \::v_unbraced \::#1#2 {
1812 \exp_after:wN \exp_arg_last_unbraced:nn
1813 \exp_after:wN {
1814 \tex_romannumeral:D -'0 \exp_eval_register:c {#2}
1815 } {#1}
1816 }

```

(End definition for `\exp_arg_last_unbraced:nn`.)

```

\exp_last_unbraced:NV
\exp_last_unbraced:No
\exp_last_unbraced:Nv
\exp_last_unbraced:Nf
\exp_last_unbraced:NcV
\exp_last_unbraced:NNV
\exp_last_unbraced:NNo
\exp_last_unbraced:NNNV
\exp_last_unbraced:NNNo

```

Now the business end.

```

1817 \cs_new_nopar:Npn \exp_last_unbraced:Nf { \::f_unbraced \:: }
1818 \cs_new_nopar:Npn \exp_last_unbraced:NV { \::V_unbraced \:: }
1819 \cs_new_nopar:Npn \exp_last_unbraced:No { \::o_unbraced \:: }
1820 \cs_new_nopar:Npn \exp_last_unbraced:Nv { \::v_unbraced \:: }
1821 \cs_new_nopar:Npn \exp_last_unbraced:NcV {
1822 \::c \::V_unbraced \::
1823 }
1824 \cs_new_nopar:Npn \exp_last_unbraced:NNV {
1825 \::N \::V_unbraced \::
1826 }
1827 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3 {
1828 \exp_after:wN #1 \exp_after:wN #2 #3
1829 }
1830 \cs_new_nopar:Npn \exp_last_unbraced:NNNV {
1831 \::N \::N \::V_unbraced \::
1832 }
1833 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4 {
1834 \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4
1835 }

```

(End definition for `\exp_last_unbraced:NV`. This function is documented on page 32.)

96.9 Items held from earlier

```

\str_if_eq_p:Vn
\str_if_eq:VnTF
\str_if_eq_p:on
\str_if_eq:onTF
\str_if_eq_p:nV
\str_if_eq:nVTF
\str_if_eq_p:no
\str_if_eq:noTF
\str_if_eq_p:VV
\str_if_eq:VVTF

```

These cannot come earlier as they need `\cs_generate_variant:Nn`.

```

1836 \cs_generate_variant:Nn \str_if_eq_p:nn { V }
1837 \cs_generate_variant:Nn \str_if_eq_p:nn { o }
1838 \cs_generate_variant:Nn \str_if_eq_p:nn { nV }
1839 \cs_generate_variant:Nn \str_if_eq_p:nn { no }
1840 \cs_generate_variant:Nn \str_if_eq_p:nn { VV }
1841 \cs_generate_variant:Nn \str_if_eq:nnT { V }
1842 \cs_generate_variant:Nn \str_if_eq:nnT { o }
1843 \cs_generate_variant:Nn \str_if_eq:nnT { nV }
1844 \cs_generate_variant:Nn \str_if_eq:nnT { no }

```

```

1845 \cs_generate_variant:Nn \str_if_eq:nnT { VV }
1846 \cs_generate_variant:Nn \str_if_eq:nnF { V }
1847 \cs_generate_variant:Nn \str_if_eq:nnF { o }
1848 \cs_generate_variant:Nn \str_if_eq:nnF { nV }
1849 \cs_generate_variant:Nn \str_if_eq:nnF { no }
1850 \cs_generate_variant:Nn \str_if_eq:nnF { VV }
1851 \cs_generate_variant:Nn \str_if_eq:nnTF { V }
1852 \cs_generate_variant:Nn \str_if_eq:nnTF { o }
1853 \cs_generate_variant:Nn \str_if_eq:nnTF { nV }
1854 \cs_generate_variant:Nn \str_if_eq:nnTF { no }
1855 \cs_generate_variant:Nn \str_if_eq:nnTF { VV }

```

(End definition for `\str_if_eq_p:Vn`. This function is documented on page [11](#).)

```

1856 </initex | package>

```

97 l3prg implementation

The following test files are used for this code: `m3prg001.lvt`, `m3prg002.lvt`, `m3prg003.lvt`.

97.1 Variables

<code>\l_tmpa_bool</code> <code>\g_tmpa_bool</code>	Reserved booleans.
--	--------------------

`\g_prg_inline_level_int` Global variable to track the nesting of the stepwise inline loop.

97.2 Module code

We start by ensuring that the required packages are loaded.

```

1857 <*package>
1858 \ProvidesExplPackage
1859   {\filename}{\filedate}{\fileversion}{\filedescription}
1860 \package_check_loaded_expl:
1861 </package>
1862 <*initex | package>

```

These are all defined in `l3basics`, as they are needed “early”. This is just a reminder that that is the case!

(End definition for `\prg_return_true:`. This function is documented on page [34](#).)

```

\prg_return_true:
\prg_return_false:
\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
  \prg_set_protected_conditional:Npnn
  \prg_new_protected_conditional:Npnn
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
  \prg_set_protected_conditional:Nnn
  \prg_new_protected_conditional:Nnn
\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn

```

97.3 Choosing modes

`\mode_if_vertical_p:` For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\c_zero` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

`\mode_if_vertical:TF`

```
1863 \prg_set_conditional:Npnn \mode_if_vertical: {p,TF,T,F}{
1864   \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi:
1865 }
```

(End definition for `\mode_if_vertical:`. These functions are documented on page 39.)

`\mode_if_horizontal_p:` For testing horizontal mode.

`\mode_if_horizontal:TF`

```
1866 \prg_set_conditional:Npnn \mode_if_horizontal: {p,TF,T,F}{
1867   \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi:
1868 }
```

(End definition for `\mode_if_horizontal:`. These functions are documented on page 39.)

`\mode_if_inner_p:` For testing inner mode.

`\mode_if_inner:TF`

```
1869 \prg_set_conditional:Npnn \mode_if_inner: {p,TF,T,F}{
1870   \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi:
1871 }
```

(End definition for `\mode_if_inner:`. These functions are documented on page 39.)

`\mode_if_math_p:` For testing math mode. Uses the kern-save `\scan_align_safe_stop:`.

`\mode_if_math:TF`

```
1872 \prg_set_conditional:Npnn \mode_if_math: {p,TF,T,F}{
1873   \scan_align_safe_stop: \if_mode_math:
1874   \prg_return_true: \else: \prg_return_false: \fi:
1875 }
```

(End definition for `\mode_if_math:`. These functions are documented on page 39.)

Alignment safe grouping and scanning

`\group_align_safe_begin:` \TeX 's alignment structures present many problems. As Knuth says himself in *TEX: The Program*: “It’s sort of a miracle whenever `\halign` or `\valign` work, [...]” One problem relates to commands that internally issues a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we will get some sort of weird error message because the underlying `\tex_futurelet:D` will store the token at the end of the alignment template. This could be a `&_4` giving a message like `! Misplaced \cr.` or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special

group so that T_EX still thinks it's on safe ground but at the same time we don't want to introduce any brace group that may find its way to the output. The following functions help with this by using code documented only in Appendix D of *The T_EXbook*...

```

1876 \cs_new_nopar:Npn \group_align_safe_begin: {
1877   \if_false:{\fi:\if_num:w'=\c_zero\fi:}
1878 \cs_new_nopar:Npn \group_align_safe_end:   {\if_num:w'=\c_zero\fi:}

```

(End definition for `\group_align_safe_begin:` and `\group_align_safe_end:`. These functions are documented on page 40.)

`\scan_align_safe_stop:` When T_EX is in the beginning of an align cell (right after the `\cr`) it is in a somewhat strange mode as it is looking ahead to find an `\tex_omit:D` or `\tex_noalign:D` and hasn't looked at the preamble yet. Thus an `\tex_ifmode:D` test will always fail unless we insert `\scan_stop:` to stop T_EX's scanning ahead. On the other hand we don't want to insert a `\scan_stop:` every time as that will destroy kerning between letters⁹ Unfortunately there is no way to detect if we're in the beginning of an alignment cell as they have different characteristics depending on column number etc. However we *can* detect if we're in an alignment cell by checking the current group type and we can also check if the previous node was a character or ligature. What is done here is that `\scan_stop:` is only inserted iff a) we're in the outer part of an alignment cell and b) the last node *wasn't* a char node or a ligature node.

```

1879 \cs_new_nopar:Npn \scan_align_safe_stop: {
1880   \int_compare:nNnT \etex_currentgrouptype:D = \c_six
1881   {
1882     \int_compare:nNnF \etex_lastnodetype:D = \c_zero
1883     {
1884       \int_compare:nNnF \etex_lastnodetype:D = \c_seven
1885       \scan_stop:
1886     }
1887   }
1888 }

```

(End definition for `\scan_align_safe_stop:`. This function is documented on page 40.)

97.4 Producing n copies

```

\prg_replicate:nn
\prg_replicate_aux:N
\prg_replicate_first_aux:N

```

This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input 25 result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed

⁹Unless we enforce an extra pass with an appropriate value of `\pretolerance`.

down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. Finally we must ensure that the cascade comes to a peaceful end so we make it so that the original csname `TeX` is creating is simply `\prg_do_nothing:` expanding to nothing.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it will severely affect the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use. An alternative approach is to create a string of `m`'s with `\int_to_roman:w` which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```

1889 \cs_new_nopar:Npn \prg_replicate:nn #1{
1890   \cs:w prg_do_nothing:
1891   \exp_after:wN\prg_replicate_first_aux:N
1892   \tex_romannumeral:D -'\q \int_eval:n{#1} \cs_end:
1893   \cs_end:
1894 }
1895 \cs_new_nopar:Npn \prg_replicate_aux:N#1{
1896   \cs:w prg_replicate_#1:n\prg_replicate_aux:N
1897 }
1898 \cs_new_nopar:Npn \prg_replicate_first_aux:N#1{
1899   \cs:w prg_replicate_first_#1:n\prg_replicate_aux:N
1900 }

```

Then comes all the functions that do the hard work of inserting all the copies.

```

1901 \cs_new_nopar:Npn      \prg_replicate_ :n #1{}% no, this is not a typo!
1902 \cs_new:cpn {prg_replicate_0:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1903 \cs_new:cpn {prg_replicate_1:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1904 \cs_new:cpn {prg_replicate_2:n}#1{\cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1905 \cs_new:cpn {prg_replicate_3:n}#1{
1906   \cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1907 \cs_new:cpn {prg_replicate_4:n}#1{
1908   \cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1909 \cs_new:cpn {prg_replicate_5:n}#1{
1910   \cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1911 \cs_new:cpn {prg_replicate_6:n}#1{
1912   \cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1913 \cs_new:cpn {prg_replicate_7:n}#1{
1914   \cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1915 \cs_new:cpn {prg_replicate_8:n}#1{
1916   \cs_end:{#1#1#1#1#1#1#1#1#1#1}}
1917 \cs_new:cpn {prg_replicate_9:n}#1{
1918   \cs_end:{#1#1#1#1#1#1#1#1#1#1}}

```

Users shouldn't ask for something to be replicated once or even not at all but...


```

1919 \cs_new:cpn {prg_replicate_first_~:n}#1{\cs_end: \ERROR }
1920 \cs_new:cpn {prg_replicate_first_0:n}#1{\cs_end: }
1921 \cs_new:cpn {prg_replicate_first_1:n}#1{\cs_end: #1}
1922 \cs_new:cpn {prg_replicate_first_2:n}#1{\cs_end: #1#1}
1923 \cs_new:cpn {prg_replicate_first_3:n}#1{\cs_end: #1#1#1}
1924 \cs_new:cpn {prg_replicate_first_4:n}#1{\cs_end: #1#1#1#1}
1925 \cs_new:cpn {prg_replicate_first_5:n}#1{\cs_end: #1#1#1#1#1}
1926 \cs_new:cpn {prg_replicate_first_6:n}#1{\cs_end: #1#1#1#1#1#1}
1927 \cs_new:cpn {prg_replicate_first_7:n}#1{\cs_end: #1#1#1#1#1#1#1}
1928 \cs_new:cpn {prg_replicate_first_8:n}#1{\cs_end: #1#1#1#1#1#1#1#1}
1929 \cs_new:cpn {prg_replicate_first_9:n}#1{\cs_end: #1#1#1#1#1#1#1#1#1}

```

(End definition for `\prg_replicate:nn`. This function is documented on page 40.)

`\prg_stepwise_function:nnnN`

A stepwise function. Firstly we check the direction of the steps #2 since that will depend on which test we should use. If the step is positive we use a greater than test, otherwise a less than test. If the test comes out true exit, otherwise perform #4, add the step to #1 and try again with this new value of #1.

`\prg_stepwise_function_incr:nnnN`
`\prg_stepwise_function_decr:nnnN`

```

1930 \cs_new:Npn \prg_stepwise_function:nnnN #1#2{
1931   \int_compare:nNnTF{#2}<\c_zero
1932     {\exp_args:Nf\prg_stepwise_function_decr:nnnN }
1933     {\exp_args:Nf\prg_stepwise_function_incr:nnnN }
1934     {\int_eval:n{#1}}{#2}
1935 }
1936 \cs_new:Npn \prg_stepwise_function_incr:nnnN #1#2#3#4{
1937   \int_compare:nNnF {#1}>{#3}
1938   {
1939     #4{#1}
1940     \exp_args:Nf \prg_stepwise_function_incr:nnnN
1941     {\int_eval:n{#1 + #2}}
1942     {#2}{#3}{#4}
1943   }
1944 }
1945 \cs_new:Npn \prg_stepwise_function_decr:nnnN #1#2#3#4{
1946   \int_compare:nNnF {#1}<{#3}
1947   {
1948     #4{#1}
1949     \exp_args:Nf \prg_stepwise_function_decr:nnnN
1950     {\int_eval:n{#1 + #2}}
1951     {#2}{#3}{#4}
1952   }
1953 }

```

(End definition for `\prg_stepwise_function:nnnN`. This function is documented on page 40.)

`\prg_stepwise_inline:nnnn`

`\g_prg_inline_level_int`

`\prg_stepwise_inline_decr:nnnn`
`\prg_stepwise_inline_incr:nnnn`

This function uses the same approach as for instance `\clist_map_inline:Nn` to allow arbitrary nesting. First construct the special function and then call an auxiliary one which just carries the newly constructed csname. Must make assignments global when we maintain our own stack.

```

1954 \cs_new_protected:Npn \prg_stepwise_inline:nnnn #1#2#3#4{
1955   \int_gincr:N \g_prg_inline_level_int
1956   \cs_gset_nopar:cpn{prg_stepwise_inline\_int\_use:N\g_prg_inline_level_int :n}##1{#4}
1957   \int_compare:nNnTF {#2}<\c_zero
1958   {\exp_args:Ncf \prg_stepwise_inline_decr:Nnnn }
1959   {\exp_args:Ncf \prg_stepwise_inline_incr:Nnnn }
1960   {prg_stepwise_inline\_int\_use:N\g_prg_inline_level_int :n}
1961   {\int_eval:n{#1}} {#2} {#3}
1962   \int_gdecr:N \g_prg_inline_level_int
1963 }
1964 \cs_new:Npn \prg_stepwise_inline_incr:Nnnn #1#2#3#4{
1965   \int_compare:nNnF {#2}>{#4}
1966   {
1967     #1{#2}
1968     \exp_args:NNf \prg_stepwise_inline_incr:Nnnn #1
1969     {\int_eval:n{#2 + #3}} {#3}{#4}
1970   }
1971 }
1972 \cs_new:Npn \prg_stepwise_inline_decr:Nnnn #1#2#3#4{
1973   \int_compare:nNnF {#2}<{#4}
1974   {
1975     #1{#2}
1976     \exp_args:NNf \prg_stepwise_inline_decr:Nnnn #1
1977     {\int_eval:n{#2 + #3}} {#3}{#4}
1978   }
1979 }

```

(End definition for \prg_stepwise_inline:nnnn. This function is documented on page ??.)

\prg_stepwise_variable:nnnNn

Almost the same as above. Just store the value in #4 and execute #5.

```

\prg_stepwise_variable_decr:nnnNn
\prg_stepwise_variable_incr:nnnNn
1980 \cs_new_protected:Npn \prg_stepwise_variable:nnnNn #1#2 {
1981   \int_compare:nNnTF {#2}<\c_zero
1982   {\exp_args:Nf\prg_stepwise_variable_decr:nnnNn}
1983   {\exp_args:Nf\prg_stepwise_variable_incr:nnnNn}
1984   {\int_eval:n{#1}}{#2}
1985 }
1986 \cs_new_protected:Npn \prg_stepwise_variable_incr:nnnNn #1#2#3#4#5 {
1987   \int_compare:nNnF {#1}>{#3}
1988   {
1989     \cs_set_nopar:Npn #4{#1} #5
1990     \exp_args:Nf \prg_stepwise_variable_incr:nnnNn
1991     {\int_eval:n{#1 + #2}}{#2}{#3}{#4}{#5}
1992   }
1993 }
1994 \cs_new_protected:Npn \prg_stepwise_variable_decr:nnnNn #1#2#3#4#5 {
1995   \int_compare:nNnF {#1}<{#3}
1996   {
1997     \cs_set_nopar:Npn #4{#1} #5
1998     \exp_args:Nf \prg_stepwise_variable_decr:nnnNn

```

```

1999     {\int_eval:n{#1 + #2}}{#2}{#3}{#4}{#5}
2000   }
2001 }

```

(End definition for `\prg_stepwise_variable:nnnn`. This function is documented on page 41.)

97.5 Booleans

For normal booleans we set them to either `\c_true_bool` or `\c_false_bool` and then use `\if_bool:N` to choose the right branch. The functions return either the TF, T, or F case *after* ending the `\if_bool:N`. We only define the N versions here as the c versions can easily be constructed with the expansion module.

```

\bool_new:N      Defining and setting a boolean is easy.
\bool_new:c
\bool_set_true:N
\bool_set_true:c
\bool_set_false:N
\bool_set_false:c
\bool_gset_true:N
\bool_gset_true:c
\bool_gset_false:N
\bool_gset_false:c
2002 \cs_new_protected_nopar:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
2003 \cs_new_protected_nopar:Npn \bool_new:c #1 { \cs_new_eq:cN {#1} \c_false_bool }
2004 \cs_new_protected_nopar:Npn \bool_set_true:N #1 { \cs_set_eq:NN #1 \c_true_bool }
2005 \cs_new_protected_nopar:Npn \bool_set_true:c #1 { \cs_set_eq:cN {#1} \c_true_bool }
2006 \cs_new_protected_nopar:Npn \bool_set_false:N #1 { \cs_set_eq:NN #1 \c_false_bool }
2007 \cs_new_protected_nopar:Npn \bool_set_false:c #1 { \cs_set_eq:cN {#1} \c_false_bool }
2008 \cs_new_protected_nopar:Npn \bool_gset_true:N #1 { \cs_gset_eq:NN #1 \c_true_bool }
2009 \cs_new_protected_nopar:Npn \bool_gset_true:c #1 { \cs_gset_eq:cN {#1} \c_true_bool }
2010 \cs_new_protected_nopar:Npn \bool_gset_false:N #1 { \cs_gset_eq:NN #1 \c_false_bool }
2011 \cs_new_protected_nopar:Npn \bool_gset_false:c #1 { \cs_gset_eq:cN {#1} \c_false_bool }

```

(End definition for `\bool_new:N` and others. These functions are documented on page 35.)

```

\bool_set_eq:NN  Setting a boolean to another is also pretty easy.
\bool_set_eq:Nc
\bool_set_eq:cN
\bool_set_eq:cc
\bool_gset_eq:NN
\bool_gset_eq:Nc
\bool_gset_eq:cN
\bool_gset_eq:cc
2012 \cs_new_eq:NN \bool_set_eq:NN \cs_set_eq:NN
2013 \cs_new_eq:NN \bool_set_eq:Nc \cs_set_eq:Nc
2014 \cs_new_eq:NN \bool_set_eq:cN \cs_set_eq:cN
2015 \cs_new_eq:NN \bool_set_eq:cc \cs_set_eq:cc
2016 \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
2017 \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
2018 \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
2019 \cs_new_eq:NN \bool_gset_eq:cc \cs_gset_eq:cc

```

(End definition for `\bool_set_eq:NN` and others. These functions are documented on page 35.)

```

\l_tmpa_bool    A few booleans just if you need them.
\g_tmpa_bool

```

```

2020 \bool_new:N \l_tmpa_bool
2021 \bool_new:N \g_tmpa_bool

```

```

\bool_if_p:N     Straight forward here. We could optimize here if we wanted to as the boolean can just
\bool_if_p:c     be input directly.
\bool_if:N $\underline{TF}$ 
\bool_if:c $\underline{TF}$ 

```

```

2022 \prg_set_conditional:Npnn \bool_if:N #1 {p,TF,T,F}{
2023   \if_bool:N #1 \prg_return_true: \else: \prg_return_false: \fi:
2024 }
2025 \cs_generate_variant:Nn \bool_if_p:N {c}
2026 \cs_generate_variant:Nn \bool_if:NTF {c}
2027 \cs_generate_variant:Nn \bool_if:NT {c}
2028 \cs_generate_variant:Nn \bool_if:NF {c}

```

(End definition for `\bool_if:N` and `\bool_if:c`. These functions are documented on page 36.)

`\bool_while_do:Nn` A **while** loop where the boolean is tested before executing the statement. The ‘while’ version executes the code as long as the boolean is true; the ‘until’ version executes the code as long as the boolean is false.

`\bool_while_do:cn`
`\bool_until_do:Nn`
`\bool_until_do:cn`

```

2029 \cs_new:Npn \bool_while_do:Nn #1 #2 {
2030   \bool_if:NT #1 {#2 \bool_while_do:Nn #1 {#2}}
2031 }
2032 \cs_generate_variant:Nn \bool_while_do:Nn {c}

2033 \cs_new:Npn \bool_until_do:Nn #1 #2 {
2034   \bool_if:NF #1 {#2 \bool_until_do:Nn #1 {#2}}
2035 }
2036 \cs_generate_variant:Nn \bool_until_do:Nn {c}

```

(End definition for `\bool_while_do:Nn` and others. These functions are documented on page 36.)

`\bool_do_while:Nn` A **do-while** loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

`\bool_do_while:cn`
`\bool_do_until:Nn`
`\bool_do_until:cn`

```

2037 \cs_new:Npn \bool_do_while:Nn #1 #2 {
2038   #2 \bool_if:NT #1 {\bool_do_while:Nn #1 {#2}}
2039 }
2040 \cs_generate_variant:Nn \bool_do_while:Nn {c}

2041 \cs_new:Npn \bool_do_until:Nn #1 #2 {
2042   #2 \bool_if:NF #1 {\bool_do_until:Nn #1 {#2}}
2043 }
2044 \cs_generate_variant:Nn \bool_do_until:Nn {c}

```

(End definition for `\bool_do_while:Nn` and others. These functions are documented on page 36.)

97.6 Parsing boolean expressions

`\bool_if_p:n` Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with (and) for grouping, ! for logical ‘Not’, && for logical ‘And’ and || for logical **Or**. We shall use the terms Not, And, Or, Open and Close for these operations.

`\bool_if:nTF`
`\bool_get_next:N`
`\bool_cleanup:N`
`\bool_choose:NN`

`\bool_!:w`
`\bool_Not:w`
`\bool_Not:w`
`\bool_(w`
`\bool_p:w`
`\bool_8_1:w`
`\bool_I_1:w`
`\bool_8_0:w`
`\bool_I_0:w`
`\bool_)_0:w`
`\bool_)_1:w`

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a `GetNext` function:

- If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.
- If a Not is seen, insert a negating function (if-even in this case) and call GetNext.
- If none of the above, start evaluating a new expression by reinserting the token found (this is supposed to be a predicate function) in front of Eval.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

$\langle true \rangle$ And Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle false \rangle$ And Current truth value is false, logical And seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return $\langle false \rangle$.

$\langle true \rangle$ Or Current truth value is true, logical Or seen, stop evaluating the predicates within this sub-expression and break to the nearest Close. Then return $\langle true \rangle$.

$\langle false \rangle$ Or Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

$\langle true \rangle$ Close Current truth value is true, Close seen, return $\langle true \rangle$.

$\langle false \rangle$ Close Current truth value is false, Close seen, return $\langle false \rangle$.

We introduce an additional Stop operation with the following semantics:

$\langle true \rangle$ Stop Current truth value is true, return $\langle true \rangle$.

$\langle false \rangle$ Stop Current truth value is false, return $\langle false \rangle$.

The reasons for this follow below.

Now for how these works in practice. The canonical true and false values have numerical values 1 and 0 respectively. We evaluate this using the primitive `\tex_number:D` operation. First we issue a `\group_align_safe_begin:` as we are using `&&` as syntax shorthand for the And operation and we need to hide it for T_EX. We also need to finish this special group before finally returning a `\c_true_bool` or `\c_false_bool` as there might otherwise be something left in front in the input stream. For this we call the Stop operation, denoted simply by a `S` following the last Close operation.

```

2045 \cs_new:Npn \bool_if_p:n #1{
2046   \group_align_safe_begin:
2047   \bool_get_next:N ( #1 )S
2048 }
```

The GetNext operation. We make it a switch: If not a ! or (, we assume it is a predicate.

```

2049 \cs_new:Npn \bool_get_next:N #1{
2050   \use:c {
2051     bool_
2052     \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
2053     :w
2054   } #1
2055 }

```

This variant gets called when a NOT has just been entered. It (eventually) results in a reversal of the logic of the directly following material.

```

2056 \cs_new:Npn \bool_get_not_next:N #1{
2057   \use:c {
2058     bool_not_
2059     \if_meaning:w !#1 ! \else: \if_meaning:w (#1 ( \else: p \fi: \fi:
2060     :w
2061   } #1
2062 }

```

We need these later on to nullify the unity operation !!.

```

2063 \cs_new:Npn \bool_get_next:NN #1#2{
2064   \bool_get_next:N #2
2065 }
2066 \cs_new:Npn \bool_get_not_next:NN #1#2{
2067   \bool_get_not_next:N #2
2068 }

```

The Not operation. Discard the token read and reverse the truth value of the next expression if there are brackets; otherwise if we're coming up to a ! then we don't need to reverse anything (but we then want to continue scanning ahead in case some fool has written !(...)); otherwise we have a boolean that we can reverse here and now.

```

2069 \cs_new:cpn { bool_!:w } #1#2 {
2070   \if_meaning:w ( #2
2071     \exp_after:wN \bool_Not:w
2072   \else:
2073     \if_meaning:w ! #2
2074     \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_next:NN
2075   \else:
2076     \exp_after:wN \exp_after:wN \exp_after:wN \bool_Not:N
2077   \fi:
2078   \fi:
2079   #2
2080 }

```

Variant called when already inside a NOT. Essentially the opposite of the above.

```

2081 \cs_new:cpn { bool_not_!:w } #1#2 {

```

```

2082 \if_meaning:w ( #2
2083   \exp_after:wN \bool_not_Not:w
2084 \else:
2085   \if_meaning:w ! #2
2086   \exp_after:wN \exp_after:wN \exp_after:wN \bool_get_not_next:NN
2087 \else:
2088   \exp_after:wN \exp_after:wN \exp_after:wN \bool_not_Not:N
2089 \fi:
2090 \fi:
2091 #2
2092 }

```

These occur when processing `!(...)`. The idea is to use a variant of `\bool_get_next:N` that finishes its parsing with a logic reversal. Of course, the double logic reversal gets us back to where we started.

```

2093 \cs_new:Npn \bool_Not:w {
2094   \exp_after:wN \tex_number:D \bool_get_not_next:N
2095 }
2096 \cs_new:Npn \bool_not_Not:w {
2097   \exp_after:wN \tex_number:D \bool_get_next:N
2098 }

```

These occur when processing `!<bool>` and can be evaluated directly.

```

2099 \cs_new:Npn \bool_Not:N #1 {
2100   \exp_after:wN \bool_p:w
2101   \if_meaning:w #1 \c_true_bool
2102     \c_false_bool
2103 \else:
2104   \c_true_bool
2105 \fi:
2106 }
2107 \cs_new:Npn \bool_not_Not:N #1 {
2108   \exp_after:wN \bool_p:w
2109   \if_meaning:w #1 \c_true_bool
2110     \c_true_bool
2111 \else:
2112   \c_false_bool
2113 \fi:
2114 }

```

The Open operation. Discard the token read and start a sub-expression. `\bool_get_next:N` continues building up the logical expressions as usual; `\bool_not_cleanup:N` is what reverses the logic if we're inside `!(...)`.

```

2115 \cs_new:cpn {bool_(~w)#1{
2116   \exp_after:wN \bool_cleanup:N \tex_number:D \bool_get_next:N
2117 }
2118 \cs_new:cpn {bool_not_(~w)#1{
2119   \exp_after:wN \bool_not_cleanup:N \tex_number:D \bool_get_next:N

```

```
2120 }
```

Otherwise just evaluate the predicate and look for And, Or or Close afterward.

```
2121 \cs_new:cpn {bool_p:w}{\exp_after:wN \bool_cleanup:N \tex_number:D }
2122 \cs_new:cpn {bool_not_p:w}{\exp_after:wN \bool_not_cleanup:N \tex_number:D }
```

This cleanup function can be omitted once predicates return their true/false booleans outside the conditionals.

```
2123 \cs_new_nopar:Npn \bool_cleanup:N #1{
2124   \exp_after:wN \bool_choose:NN \exp_after:wN #1
2125   \int_to_roman:w-'\q
2126 }
2127 \cs_new_nopar:Npn \bool_not_cleanup:N #1{
2128   \exp_after:wN \bool_not_choose:NN \exp_after:wN #1
2129   \int_to_roman:w-'\q
2130 }
```

Branching the six way switch. Reversals should be reasonably straightforward. When programming this, however, I got things around the wrong way a few times. (Will's hacks onto Morten's code, that is.)

```
2131 \cs_new_nopar:Npn \bool_choose:NN #1#2{ \use:c{bool_#2_#1:w} }
2132 \cs_new_nopar:Npn \bool_not_choose:NN #1#2{ \use:c{bool_not_#2_#1:w} }
```

Continues scanning. Must remove the second & or |.

```
2133 \cs_new_nopar:cpn{bool_&_1:w}{\bool_get_next:N}
2134 \cs_new_nopar:cpn{bool_|_0:w}{\bool_get_next:N}
2135 \cs_new_nopar:cpn{bool_not_&_0:w}{\bool_get_next:N}
2136 \cs_new_nopar:cpn{bool_not_|_1:w}{\bool_get_next:N}
```

Closing a group is just about returning the result. The Stop operation is similar except it closes the special alignment group before returning the boolean.

```
2137 \cs_new_nopar:cpn{bool_)_0:w}{ \c_false_bool }
2138 \cs_new_nopar:cpn{bool_)_1:w}{ \c_true_bool }
2139 \cs_new_nopar:cpn{bool_not_)_0:w}{ \c_true_bool }
2140 \cs_new_nopar:cpn{bool_not_)_1:w}{ \c_false_bool }
2141 \cs_new_nopar:cpn{bool_S_0:w}{\group_align_safe_end: \c_false_bool }
2142 \cs_new_nopar:cpn{bool_S_1:w}{\group_align_safe_end: \c_true_bool }
```

When the truth value has already been decided, we have to throw away the remainder of the current group as we are doing minimal evaluation. This is slightly tricky as there are no braces so we have to play match the () manually.

```
2143 \cs_new:cpn{bool_&_0:w}{\bool_eval_skip_to_end:Nw \c_false_bool}
2144 \cs_new:cpn{bool_|_1:w}{\bool_eval_skip_to_end:Nw \c_true_bool}
2145 \cs_new:cpn{bool_not_&_1:w}{\bool_eval_skip_to_end:Nw \c_false_bool}
2146 \cs_new:cpn{bool_not_|_0:w}{\bool_eval_skip_to_end:Nw \c_true_bool}
```


(End definition for \bool_if:n. These functions are documented on page 37.)

```
\bool_eval_skip_to_end:Nw  
  \bool_eval_skip_to_end_aux:Nw  
  \bool_eval_skip_to_end_auxii:Nw
```

There is always at least one `)` waiting, namely the outer one. However, we are facing the problem that there may be more than one that need to be finished off and we have to detect the correct number of them. Here is a complicated example showing how this is done. After evaluating the following, we realize we must skip everything after the first `And`. Note the extra `Close` at the end.

```
\c_false_bool  && ((abc) && xyz) && ((xyz) && (def)))
```

First read up to the first `Close`. This gives us the list we first read up until the first right parenthesis so we are looking at the token list

```
((abc
```

This contains two `Open` markers so we must remove two groups. Since no evaluation of the contents is to be carried out, it doesn't matter how we remove the groups as long as we wind up with the correct result. We therefore first remove a `()` pair and what preceded the `Open` – but leave the contents as it may contain `Open` tokens itself – leaving

```
(abc && xyz) && ((xyz) && (def)))
```

Another round of this gives us

```
(abc && xyz
```

which still contains an `Open` so we remove another `()` pair, giving us

```
abc && xyz && ((xyz) && (def)))
```

Again we read up to a `Close` and again find `Open` tokens:

```
abc && xyz && ((xyz
```

Further reduction gives us

```
(xyz && (def)))
```

and then

```
(xyz && (def
```

with reduction to

```
xyz && (def))
```

and ultimately we arrive at no Open tokens being skipped and we can finally close the group nicely.

This whole operation could be made a lot simpler if we were allowed to do simple pattern matching. With a new enough pdfTeX one can do that sort of thing to test for existence of particular tokens.

```
2147 \cs_new:Npn \bool_eval_skip_to_end:Nw #1#2){
2148   \bool_eval_skip_to_end_aux:Nw #1 #2(\q_no_value\q_stop{#2}
2149 }
```

If no right parenthesis, then #3 is no_value and we are done, return the boolean #1. If there is, we need to grab a () pair and then recurse

```
2150 \cs_new:Npn \bool_eval_skip_to_end_aux:Nw #1#2(#3#4\q_stop#5{
2151   \quark_if_no_value:NTF #3
2152   { #1 }
2153   { \bool_eval_skip_to_end_auxii:Nw #1 #5 }
2154 }
```

keep the boolean, throw away anything up to the (as it is irrelevant, remove a () pair but remember to reinsert #3 as it may contain (tokens!

```
2155 \cs_new:Npn \bool_eval_skip_to_end_auxii:Nw #1#2(#3){
2156   \bool_eval_skip_to_end:Nw #1#3 )
2157 }
```

(End definition for \bool_eval_skip_to_end:Nw, \bool_eval_skip_to_end_aux:Nw, and \bool_eval_skip_to_end_auxii:Nw.)

\bool_set:Nn
\bool_set:cn
\bool_gset:Nn
\bool_gset:cn

This function evaluates a boolean expression and assigns the first argument the meaning \c_true_bool or \c_false_bool.

```
2158 \cs_new:Npn \bool_set:Nn #1#2 {\tex_chardef:D #1 = \bool_if_p:n {#2}}
2159 \cs_new:Npn \bool_gset:Nn #1#2 {
2160   \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
2161 }
2162 \cs_generate_variant:Nn \bool_set:Nn {c}
2163 \cs_generate_variant:Nn \bool_gset:Nn {c}
```

(End definition for \bool_set:Nn and others. These functions are documented on page 37.)

\bool_not_p:n

The not variant just reverses the outcome of \bool_if_p:n. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```
2164 \cs_new:Npn \bool_not_p:n #1{ \bool_if_p:n{!(#1)} }
```

(End definition for \bool_not_p:n. This function is documented on page 37.)

`\bool_xor_p:nn` Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```

2165 \cs_new:Npn \bool_xor_p:nn #1#2 {
2166   \int_compare:nNnTF {\bool_if_p:n { #1 }} = {\bool_if_p:n { #2 }}
2167   {\c_false_bool}{\c_true_bool}
2168 }

```

(End definition for `\bool_xor_p:nn`. This function is documented on page 37.)

```

2169 \prg_set_conditional:Npnn \bool_if:n #1 {TF,T,F}{
2170   \if_predicate:w \bool_if_p:n{#1}
2171   \prg_return_true: \else: \prg_return_false: \fi:
2172 }

```

`\bool_while_do:nn` #1 : Predicate test
`\bool_do_while:nn` #2 : Code to execute
`\bool_until_do:nn`
`\bool_do_until:nn`

```

2173 \cs_new:Npn \bool_while_do:nn #1#2 {
2174   \bool_if:nT {#1} { #2 \bool_while_do:nn {#1}{#2} }
2175 }
2176 \cs_new:Npn \bool_until_do:nn #1#2 {
2177   \bool_if:nF {#1} { #2 \bool_until_do:nn {#1}{#2} }
2178 }
2179 \cs_new:Npn \bool_do_while:nn #1#2 {
2180   #2 \bool_if:nT {#1} { \bool_do_while:nn {#1}{#2} }
2181 }
2182 \cs_new:Npn \bool_do_until:nn #1#2 {
2183   #2 \bool_if:nF {#1} { \bool_do_until:nn {#1}{#2} }
2184 }

```

(End definition for `\bool_while_do:nn` and `\bool_do_while:nn`. These functions are documented on page 39.)

97.7 Case switch

`\prg_case_int:nnn` This case switch is in reality quite simple. It takes three arguments:
`\prg_case_int_aux:nnn`

1. An integer expression you wish to find.
2. A list of pairs of $\{\langle integer\ expr \rangle\} \{\langle code \rangle\}$. The list can be as long as is desired and $\langle integer\ expr \rangle$ can be negative.
3. The code to be executed if the value wasn't found.

We don't need the else case here yet, so leave it dangling in the input stream.

```

2185 \cs_new:Npn \prg_case_int:nnn #1 #2 {

```

We will be parsing on #1 for each step so we might as well evaluate it first in case it is complicated.

```
2186 \exp_args:Nf \prg_case_int_aux:nnn { \int_eval:n{#1}} #2
```

The ? below is just so there are enough arguments when we reach the end. And it made you look. ;-)

```
2187 \q_recursion_tail ? \q_recursion_stop
2188 }
2189 \cs_new:Npn \prg_case_int_aux:nnn #1#2#3{
```

If we reach the end, return the else case. We just remove braces.

```
2190 \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
```

Otherwise we compare (which evaluates #2 for us)

```
2191 \int_compare:nNnTF{#1}={#2}
```

If true, we want to remove the remainder of the list, the else case and then execute the code specified. \prg_end_case:nw {#3} does just that in one go. This means f style expansion works the way one wants it to work.

```
2192 { \prg_end_case:nw {#3} }
2193 { \prg_case_int_aux:nnn {#1}}
2194 }
```

(End definition for \prg_case_int:nnn. This function is documented on page 38.)

\prg_case_dim:nnn
\prg_case_dim_aux:nnn

Same as \prg_case_dim:nnn except it is for *<dim>* registers.

```
2195 \cs_new:Npn \prg_case_dim:nnn #1 #2 {
2196 \exp_args:No \prg_case_dim_aux:nnn {\dim_eval:n{#1}} #2
2197 \q_recursion_tail ? \q_recursion_stop
2198 }
2199 \cs_new:Npn \prg_case_dim_aux:nnn #1#2#3{
2200 \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
2201 \dim_compare:nNnTF{#1}={#2}
2202 { \prg_end_case:nw {#3} }
2203 { \prg_case_dim_aux:nnn {#1}}
2204 }
```

(End definition for \prg_case_dim:nnn. This function is documented on page 38.)

\prg_case_str:nnn
\prg_case_str_aux:nnn

Same as \prg_case_dim:nnn except it is for strings.

```
2205 \cs_new:Npn \prg_case_str:nnn #1 #2 {
2206 \prg_case_str_aux:nnn {#1} #2
2207 \q_recursion_tail ? \q_recursion_stop
2208 }
```

```

2209 \cs_new:Npn \prg_case_str_aux:nnn #1#2#3{
2210   \quark_if_recursion_tail_stop_do:nn{#2}{\use:n}
2211   \str_if_eq:xxTF{#1}{#2}
2212   { \prg_end_case:nw {#3} }
2213   { \prg_case_str_aux:nnn {#1}}
2214 }

```

(End definition for `\prg_case_str:nnn`. This function is documented on page 38.)

`\prg_case_tl:Nnn` Same as `\prg_case_dim:nnn` except it is for token list variables.
`\prg_case_tl_aux:NNn`

```

2215 \cs_new:Npn \prg_case_tl:Nnn #1 #2 {
2216   \prg_case_tl_aux:NNn #1 #2
2217   \q_recursion_tail ? \q_recursion_stop
2218 }
2219 \cs_new:Npn \prg_case_tl_aux:NNn #1#2#3{
2220   \quark_if_recursion_tail_stop_do:Nn #2{\use:n}
2221   \tl_if_eq:NNTF #1 #2
2222   { \prg_end_case:nw {#3} }
2223   { \prg_case_tl_aux:NNn #1}
2224 }

```

(End definition for `\prg_case_tl:Nnn`. This function is documented on page 38.)

`\prg_end_case:nw` Ending a case switch is always performed the same way so we optimize for this. #1 is the code to execute, #2 the remainder, and #3 the dangling else case.

```

2225 \cs_new:Npn \prg_end_case:nw #1#2\q_recursion_stop#3{#1}

```

(End definition for `\prg_end_case:nw`.)

97.8 Sorting

`\prg_define_quicksort:nnn` #1 is the name, #2 and #3 are the tokens enclosing the argument. For the somewhat strange `<clist>` type which doesn't enclose the items but uses a separator we define it by hand afterwards. When doing the first pass, the algorithm wraps all elements in braces and then uses a generic quicksort which works on token lists.

As an example

```

\prg_define_quicksort:nnn{seq}{\seq_elt:w}{\seq_elt_end:w}

```

defines the user function `\seq_quicksort:n` and furthermore expects to use the two functions `\seq_quicksort_compare:nnTF` which compares the items and `\seq_quicksort_function:n` which is placed before each sorted item. It is up to the programmer to define these functions when needed. For the `seq` type a sequence is a token list variable, so one additionally has to define

```

\cs_set_nopar:Npn \seq_quicksort:N{\exp_args:No\seq_quicksort:n}

```

For details on the implementation see “Sorting in \TeX ’s Mouth” by Bernd Raichle. Firstly we define the function for parsing the initial list and then the braced list afterwards.

```

2226 \cs_new_protected_nopar:Npn \prg_define_quicksort:nnn #1#2#3 {
2227   \cs_set:cpx{#1_quicksort:n}##1{
2228     \exp_not:c{#1_quicksort_start_partition:w} ##1
2229     \exp_not:n{#2\q_nil#3\q_stop}
2230   }
2231   \cs_set:cpx{#1_quicksort_braced:n}##1{
2232     \exp_not:c{#1_quicksort_start_partition_braced:n} ##1
2233     \exp_not:N\q_nil\exp_not:N\q_stop
2234   }
2235   \cs_set:cpx {#1_quicksort_start_partition:w} #2 ##1 #3{
2236     \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2237     \exp_not:c{#1_quicksort_do_partition_i:nnnw} {##1}{-}{-}
2238   }
2239   \cs_set:cpx {#1_quicksort_start_partition_braced:n} ##1 {
2240     \exp_not:N \quark_if_nil:nT {##1}\exp_not:N \use_none_delimit_by_q_stop:w
2241     \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn} {##1}{-}{-}
2242   }

```

Now for doing the partitions.

```

2243 \cs_set:cpx {#1_quicksort_do_partition_i:nnnw} ##1##2##3 #2 ##4 #3 {
2244   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2245   {
2246     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2247     \exp_not:c{#1_quicksort_partition_greater_ii:nnnn}
2248     \exp_not:c{#1_quicksort_partition_less_ii:nnnn}
2249   }
2250   {##1}{##2}{##3}{##4}
2251 }
2252 \cs_set:cpx {#1_quicksort_do_partition_i_braced:nnnn} ##1##2##3##4 {
2253   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2254   {
2255     \exp_not:c{#1_quicksort_compare:nnTF}{##1}{##4}
2256     \exp_not:c{#1_quicksort_partition_greater_ii_braced:nnnn}
2257     \exp_not:c{#1_quicksort_partition_less_ii_braced:nnnn}
2258   }
2259   {##1}{##2}{##3}{##4}
2260 }
2261 \cs_set:cpx {#1_quicksort_do_partition_ii:nnnw} ##1##2##3 #2 ##4 #3 {
2262   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2263   {
2264     \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2265     \exp_not:c{#1_quicksort_partition_less_i:nnnn}
2266     \exp_not:c{#1_quicksort_partition_greater_i:nnnn}
2267   }
2268   {##1}{##2}{##3}{##4}
2269 }

```

```

2270 \cs_set:cpx {#1_quicksort_do_partition_ii_braced:nnnn} ##1##2##3##4 {
2271   \exp_not:N \quark_if_nil:nTF {##4} \exp_not:c {#1_do_quicksort_braced:nnnnw}
2272   {
2273     \exp_not:c{#1_quicksort_compare:nnTF}{##4}{##1}
2274     \exp_not:c{#1_quicksort_partition_less_i_braced:nnnn}
2275     \exp_not:c{#1_quicksort_partition_greater_i_braced:nnnn}
2276   }
2277   {##1}{##2}{##3}{##4}
2278 }

```

This part of the code handles the two branches in each sorting. Again we will also have to do it braced.

```

2279 \cs_set:cpx {#1_quicksort_partition_less_i:nnnn} ##1##2##3##4{
2280   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##2}{##4}{##3}}
2281 \cs_set:cpx {#1_quicksort_partition_less_ii:nnnn} ##1##2##3##4{
2282   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##3}{##4}}
2283 \cs_set:cpx {#1_quicksort_partition_greater_i:nnnn} ##1##2##3##4{
2284   \exp_not:c{#1_quicksort_do_partition_i:nnnw}{##1}{##4}{##2}{##3}}
2285 \cs_set:cpx {#1_quicksort_partition_greater_ii:nnnn} ##1##2##3##4{
2286   \exp_not:c{#1_quicksort_do_partition_ii:nnnw}{##1}{##2}{##4}{##3}}
2287 \cs_set:cpx {#1_quicksort_partition_less_i_braced:nnnn} ##1##2##3##4{
2288   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##2}{##4}{##3}}
2289 \cs_set:cpx {#1_quicksort_partition_less_ii_braced:nnnn} ##1##2##3##4{
2290   \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##3}{##4}}
2291 \cs_set:cpx {#1_quicksort_partition_greater_i_braced:nnnn} ##1##2##3##4{
2292   \exp_not:c{#1_quicksort_do_partition_i_braced:nnnn}{##1}{##4}{##2}{##3}}
2293 \cs_set:cpx {#1_quicksort_partition_greater_ii_braced:nnnn} ##1##2##3##4{
2294   \exp_not:c{#1_quicksort_do_partition_ii_braced:nnnn}{##1}{##2}{##4}{##3}}

```

Finally, the big kahuna! This is where the sub-lists are sorted.

```

2295 \cs_set:cpx {#1_do_quicksort_braced:nnnnw} ##1##2##3##4\q_stop {
2296   \exp_not:c{#1_quicksort_braced:n}{##2}
2297   \exp_not:c{#1_quicksort_function:n}{##1}
2298   \exp_not:c{#1_quicksort_braced:n}{##3}
2299 }
2300 }

```

(End definition for \prg_define_quicksort:nnn.)

\prg_quicksort:n A simple version. Sorts a list of tokens, uses the function `\prg_quicksort_compare:nnTF` to compare items, and places the function `\prg_quicksort_function:n` in front of each of them.

```

2301 \prg_define_quicksort:nnn {prg}{-}{-}

```

(End definition for \prg_quicksort:n. This function is documented on page 41.)

\prg_quicksort_function:n
\prg_quicksort_compare:nnTF

```

2302 \cs_set:Npn \prg_quicksort_function:n {\ERROR}
2303 \cs_set:Npn \prg_quicksort_compare:nnTF {\ERROR}

```

(End definition for `\prg_quicksort_function:n`. This function is documented on page 41.)

97.9 Variable type and scope

`\prg_variable_get_scope:N` Expandable functions to find the type of a variable, and to return g if the variable is global. The trick for `\prg_variable_get_scope:N` is the same as that in `\cs_split_function:NN`, but it can be simplified as the requirements here are less complex.

`\prg_variable_get_type:N`

`\prg_variable_get_type:w`

```

2304 \group_begin:
2305   \tex_lccode:D '\& = '\g \tex_relax:D
2306   \tex_catcode:D '\& = \c_twelve \tex_relax:D
2307   \tl_to_lowercase:n {
2308     \group_end:
2309     \cs_new_nopar:Npn \prg_variable_get_scope:N #1 {
2310       \exp_last_unbraced:Nf \prg_variable_get_scope_aux:w
2311       { \cs_to_str:N #1 \exp_stop_f: \q_stop }
2312     }
2313     \cs_new_nopar:Npn \prg_variable_get_scope_aux:w #1#2 \q_stop {
2314       \token_if_eq_meaning:NNT & #1 {g}
2315     }
2316   }
2317   \group_begin:
2318     \tex_lccode:D '\& = '\_ \tex_relax:D
2319     \tex_catcode:D '\& = \c_twelve \tex_relax:D
2320     \tl_to_lowercase:n {
2321       \group_end:
2322       \cs_new_nopar:Npn \prg_variable_get_type:N #1 {
2323         \exp_after:wN \p;rg_variable_get_type_aux:w
2324         \token_to_str:N #1 & a \q_stop
2325       }
2326       \cs_new_nopar:Npn \prg_variable_get_type_aux:w #1 & #2#3 \q_stop {
2327         \token_if_eq_meaning:NNTF a #2 {
2328           #1
2329         }{
2330           \prg_variable_get_type_aux:w #2#3 \q_stop
2331         }
2332       }
2333     }

```

(End definition for `\prg_variable_get_scope:N`. This function is documented on page 42.)

97.10 Mapping to variables

`\prg_new_map_functions:Nn` The idea here is to generate all of the various mapping functions in one go. Everything is done with expansion so that the performance hit is taken at definition time and not at

`\prg_set_map_functions:Nn`

point of use. The inline version uses a counter as this keeps things nestable, and global to avoid problems with, for example, table cells.

```

2334 \cs_new_protected:Npn \prg_new_map_functions:Nn #1#2 {
2335   \cs_if_free:cTF { #2 _map_function:NN }
2336   { \prg_set_map_functions:Nn #1 {#2} }
2337   {
2338     \msg_kernel_error:nxx { code } { csname-already-defined }
2339     { \token_to_str:c { #2 _map_function:NN } }
2340   }
2341 }
2342 \cs_new_protected:Npn \prg_set_map_functions:Nn #1#2 {
2343   \cs_gset_nopar:cpx { #2 _map_function:NN } ##1##2
2344   {
2345     \exp_not:N \tl_if_empty:NF ##1
2346     {
2347       \exp_not:N \exp_after:wN
2348       \exp_not:c { #2 _map_function_aux:Nw }
2349       \exp_not:N \exp_after:wN ##2 ##1
2350       \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2351     }
2352   }
2353   \cs_gset:cpx { #2 _map_function:nN } ##1##2
2354   {
2355     \exp_not:N \tl_if_blank:nF {##1}
2356     {
2357       \exp_not:c { #2 _map_function_aux:Nw } ##2 ##1
2358       \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2359     }
2360   }
2361   \cs_gset:cpx { #2 _map_function_aux:Nw } ##1##2 #1
2362   {
2363     \exp_not:N \quark_if_recursion_tail_stop:n {##2}
2364     ##1 {##2}
2365     \exp_not:c { #2 _map_function_aux:Nw } ##1
2366   }
2367   \cs_if_free:cT { g_ #2 _map_inline_int }
2368   { \int_new:c { g_ #2 _map_inline_int } }
2369   \cs_gset_protected_nopar:cpx { #2 _map_inline:Nn } ##1##2
2370   {
2371     \exp_not:N \tl_if_empty:NF ##1
2372     {
2373       \exp_not:N \int_gincr:N \exp_not:c { g_ #2 _map_inline_int }
2374       \cs_gset:cpn
2375       {
2376         #2 _map_inline_
2377         \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2378         :n
2379       }
2380       #####1 {##2}

```

```

2381 \exp_not:N \exp_last_unbraced:NcV
2382 \exp_not:c { #2 _map_function_aux:Nw }
2383 {
2384   #2 _map_inline_
2385   \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2386   :n
2387 }
2388 ##1 \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2389 \exp_not:N \int_gdecr:N \exp_not:c { g_ #2 _map_inline_int }
2390 }
2391 }
2392 \cs_gset_protected:cpx { #2 _map_inline:nn } ##1##2
2393 {
2394   \exp_not:N \tl_if_empty:nF {##1}
2395   {
2396     \exp_not:N \int_gincr:N \exp_not:c { g_ #2 _map_inline_int }
2397     \cs_gset:cpn
2398     {
2399       #2 _map_inline_
2400       \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2401       :n
2402     }
2403     #####1 {##2}
2404     \exp_not:N \exp_args:Nc
2405     \exp_not:c { #2 _map_function_aux:Nw }
2406     {
2407       #2 _map_inline_
2408       \exp_not:N \int_use:N \exp_not:c { g_ #2 _map_inline_int }
2409       :n
2410     }
2411     ##1 \exp_not:n { #1 \q_recursion_tail #1 \q_recursion_stop }
2412     \exp_not:N \int_gdecr:N \exp_not:c { g_ #2 _map_inline_int }
2413   }
2414 }
2415 \cs_gset_eq:cN { #2 _map_break: }
2416 \use_none_delimit_by_q_recursion_stop:w
2417 }

```

(End definition for `\prg_new_map_functions:Nn`. This function is documented on page 42.)

That's it (for now).

```

2418 </initex | package>

```

98 l3quark implementation

The following test files are used for this code: `m3quark001.lvt`.

We start by ensuring that the required packages are loaded. We check for `l3expan` since this is a basic package that is essential for use of any higher-level package.

```

2419 <*package>
2420 \ProvidesExplPackage
2421   {\filename}{\filedate}{\fileversion}{\filedescription}
2422 \package_check_loaded_expl:
2423 </package>
2424 <*initex | package>

```

`\quark_new:N` Allocate a new quark.

```

2425 \cs_new_protected_nopar:Npn \quark_new:N #1 { \tl_const:Nn #1 {#1} }

```

(End definition for `\quark_new:N`. This function is documented on page 43.)

`\q_stop` `\q_no_value` `\q_nil` `\q_stop` is often used as a marker in parameter text, `\q_no_value` is the canonical missing value, and `\q_nil` represents a nil pointer in some data structures.

```

2426 \quark_new:N \q_stop
2427 \quark_new:N \q_no_value
2428 \quark_new:N \q_nil

```

`\q_error` `\q_mark` We need two additional quarks. `\q_error` delimits the end of the computation for purposes of error recovery. `\q_mark` is used in parameter text when we need a scanning boundary that is distinct from `\q_stop`.

```

2429 \quark_new:N\q_error
2430 \quark_new:N\q_mark

```

`\q_recursion_tail` `\q_recursion_stop` Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. `\q_recursion_stop` is placed directly after the list.

```

2431 \quark_new:N\q_recursion_tail
2432 \quark_new:N\q_recursion_stop

```

`\quark_if_recursion_tail_stop:N` `\quark_if_recursion_tail_stop_do:Nn` When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapped up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```

2433 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
2434   {
2435     \tex_ifx:D #1 \q_recursion_tail

```

```

2436     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2437     \tex_fi:D
2438   }
2439 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1#2
2440 {
2441   \tex_ifx:D #1 \q_recursion_tail
2442   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2443   \tex_else:D
2444   \exp_after:wN \use_none:n
2445   \tex_fi:D
2446   {#2}
2447 }

```

(End definition for `\quark_if_recursion_tail_stop:N`. This function is documented on page 45.)

```

\quark_if_recursion_tail_stop:n
\quark_if_recursion_tail_stop:o
\quark_if_recursion_tail_stop_do:nn
\quark_if_recursion_tail_stop_do:on
\quark_if_recursion_tail_aux:w

```

The same idea applies when testing multiple tokens, but here a little more care is needed. It is possible that #1 might be something like `{{{a}}}` or `{ab\iffalse}\fi`, which will therefore need to be tested in a detokenized manner. The way that this is done is using `\tex_ifcat:D`, with the idea being that this test will be `true` provided the auxiliary function returns nothing at all. If the auxiliary returns anything, it will be detokenized and so the test will be both `false` and `safe`.

```

2448 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
2449 {
2450   \tex_ifcat:D
2451   A
2452   \etex_detokenize:D \exp_after:wN
2453   {
2454     \quark_if_recursion_tail_aux:w #1 \q_recursion_stop
2455     \q_recursion_tail \q_recursion_stop \q_stop
2456   }
2457   A
2458   \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2459   \tex_fi:D
2460 }
2461 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1#2
2462 {
2463   \tex_ifcat:D
2464   A
2465   \etex_detokenize:D \exp_after:wN
2466   {
2467     \quark_if_recursion_tail_aux:w #1 \q_recursion_stop
2468     \q_recursion_tail \q_recursion_stop \q_stop
2469   }
2470   A
2471   \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2472   \tex_else:D
2473   \exp_after:wN \use_none:n
2474   \tex_fi:D
2475   {#2}

```

```

2476 }
2477 \cs_new:Npn \quark_if_recursion_tail_aux:w
2478 #1 \q_recursion_tail #2 \q_recursion_stop #3 \q_stop
2479 { #1 #2 }
2480 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
2481 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:do:nn { o }

```

(End definition for `\quark_if_recursion_tail_stop:n` and `\quark_if_recursion_tail_stop:o`. These functions are documented on page 45.)

`\quark_if_no_value_p:N` Here we test if we found a special quark as the first argument. We better start with
`\quark_if_no_value_p:n` `\q_no_value` as the first argument since the whole thing may otherwise loop if #1 is
`\quark_if_no_value:NTF` wrongly given a string like aabc instead of a single token.¹⁰
`\quark_if_no_value:nTF`

```

2482 \prg_new_conditional:Nnn \quark_if_no_value:N {p,TF,T,F} {
2483   \if_meaning:w \q_no_value #1
2484   \prg_return_true: \else: \prg_return_false: \fi:
2485 }

```

These tests are easy with `\pdf_strcmp:D` available.

```

2486 \prg_new_conditional:Nnn \quark_if_no_value:n {p,TF,T,F} {
2487   \if_num:w \pdf_strcmp:D
2488     {\exp_not:N \q_no_value}
2489     {\exp_not:n{#1}} = \c_zero
2490   \prg_return_true: \else: \prg_return_false:
2491   \fi:
2492 }

```

(End definition for `\quark_if_no_value:N` and `\quark_if_no_value:n`. These functions are documented on page 43.)

`\quark_if_nil_p:N` A function to check for the presence of `\q_nil`.
`\quark_if_nil:NTF`

```

2493 \prg_new_conditional:Nnn \quark_if_nil:N {p,TF,T,F} {
2494   \if_meaning:w \q_nil #1 \prg_return_true: \else: \prg_return_false: \fi:
2495 }

```

(End definition for `\quark_if_nil:N`. These functions are documented on page 44.)

`\quark_if_nil_p:n` A function to check for the presence of `\q_nil`.

`\quark_if_nil_p:V`
`\quark_if_nil_p:o`
`\quark_if_nil:nTF`
`\quark_if_nil:VTF`
`\quark_if_nil:oTF`

```

2496 \prg_new_conditional:Nnn \quark_if_nil:n {p,TF,T,F} {
2497   \if_num:w \pdf_strcmp:D
2498     {\exp_not:N \q_nil}
2499     {\exp_not:n{#1}} = \c_zero
2500   \prg_return_true: \else: \prg_return_false:
2501   \fi:
2502 }

```

¹⁰It may still loop in special circumstances however!

```

2503 \cs_generate_variant:Nn \quark_if_nil_p:n {V}
2504 \cs_generate_variant:Nn \quark_if_nil:nTF {V}
2505 \cs_generate_variant:Nn \quark_if_nil:nT {V}
2506 \cs_generate_variant:Nn \quark_if_nil:nF {V}
2507 \cs_generate_variant:Nn \quark_if_nil_p:n {o}
2508 \cs_generate_variant:Nn \quark_if_nil:nTF {o}
2509 \cs_generate_variant:Nn \quark_if_nil:nT {o}
2510 \cs_generate_variant:Nn \quark_if_nil:nF {o}

```

(End definition for `\quark_if_nil:n`, `\quark_if_nil:V`, and `\quark_if_nil:o`. These functions are documented on page 44.)

99 l3token implementation

99.1 Documentation of internal functions

```

\l_peek_true_tl
\l_peek_false_tl

```

These token list variables are used internally when choosing either the true or false branches of a test.

```

\l_peek_search_tl

```

Used to store `\l_peek_search_token`.

```

\peek_tmp:w

```

Scratch function used to gobble tokens from the input stream.

```

\l_peek_true_aux_tl
\c_peek_true_remove_next_tl

```

These token list variables are used internally when choosing either the true or false branches of a test.

```

\peek_ignore_spaces_execute_branches:
\peek_ignore_spaces_aux:

```

Functions used to ignore space tokens in the input stream.

99.2 Module code

First a few required packages to get this going.

```

2511 <*package>

```

```

2512 \ProvidesExplPackage
2513   {\filename}{\filedate}{\fileversion}{\filedescription}
2514 \package_check_loaded_expl:
2515 </package>
2516 <*initex | package>

```

99.3 Character tokens

```

\char_set_catcode:w
\char_set_catcode:nn
\char_value_catcode:w
\char_value_catcode:n
\char_show_value_catcode:w
\char_show_value_catcode:n
2517 \cs_new_eq:NN \char_set_catcode:w \tex_catcode:D
2518 \cs_new_protected_nopar:Npn \char_set_catcode:nn #1#2 {
2519   \char_set_catcode:w #1 = \int_eval:w #2\int_eval_end:
2520 }
2521 \cs_new_nopar:Npn \char_value_catcode:w { \int_use:N \tex_catcode:D }
2522 \cs_new_nopar:Npn \char_value_catcode:n #1 {
2523   \char_value_catcode:w \int_eval:w #1\int_eval_end:
2524 }
2525 \cs_new_nopar:Npn \char_show_value_catcode:w {
2526   \tex_showthe:D \tex_catcode:D
2527 }
2528 \cs_new_nopar:Npn \char_show_value_catcode:n #1 {
2529   \char_show_value_catcode:w \int_eval:w #1\int_eval_end:
2530 }

```

(End definition for `\char_set_catcode:w` and others. These functions are documented on page 46.)

```

\char_make_escape:N
\char_make_begin_group:N
\char_make_end_group:N
\char_make_math_shift:N
\char_make_alignment:N
\char_make_end_line:N
\char_make_parameter:N
\char_make_math_superscript:N
\char_make_math_subscript:N
\char_make_ignore:N
\char_make_space:N
\char_make_letter:N
\char_make_other:N
\char_make_active:N
\char_make_comment:N
\char_make_invalid:N
2531 \cs_new_protected_nopar:Npn \char_make_escape:N #1 { \char_set_catcode:nn {'#1} {\c
2532 \cs_new_protected_nopar:Npn \char_make_begin_group:N #1 { \char_set_catcode:nn {'#1} {\c
2533 \cs_new_protected_nopar:Npn \char_make_end_group:N #1 { \char_set_catcode:nn {'#1} {\c
2534 \cs_new_protected_nopar:Npn \char_make_math_shift:N #1 { \char_set_catcode:nn {'#1} {\c
2535 \cs_new_protected_nopar:Npn \char_make_alignment:N #1 { \char_set_catcode:nn {'#1} {\c
2536 \cs_new_protected_nopar:Npn \char_make_end_line:N #1 { \char_set_catcode:nn {'#1} {\c
2537 \cs_new_protected_nopar:Npn \char_make_parameter:N #1 { \char_set_catcode:nn {'#1} {\c
2538 \cs_new_protected_nopar:Npn \char_make_math_superscript:N #1 { \char_set_catcode:nn {'#1} {\c
2539 \cs_new_protected_nopar:Npn \char_make_math_subscript:N #1 { \char_set_catcode:nn {'#1} {\c
2540 \cs_new_protected_nopar:Npn \char_make_ignore:N #1 { \char_set_catcode:nn {'#1} {\c
2541 \cs_new_protected_nopar:Npn \char_make_space:N #1 { \char_set_catcode:nn {'#1} {\c
2542 \cs_new_protected_nopar:Npn \char_make_letter:N #1 { \char_set_catcode:nn {'#1} {\c
2543 \cs_new_protected_nopar:Npn \char_make_other:N #1 { \char_set_catcode:nn {'#1} {\c
2544 \cs_new_protected_nopar:Npn \char_make_active:N #1 { \char_set_catcode:nn {'#1} {\c
2545 \cs_new_protected_nopar:Npn \char_make_comment:N #1 { \char_set_catcode:nn {'#1} {\c
2546 \cs_new_protected_nopar:Npn \char_make_invalid:N #1 { \char_set_catcode:nn {'#1} {\c

```

(End definition for `\char_make_escape:N` and others. These functions are documented on page 47.)

```

\char_make_escape:n
\char_make_begin_group:n
\char_make_end_group:n
\char_make_math_shift:n
\char_make_alignment:n
\char_make_end_line:n
\char_make_parameter:n
\char_make_math_superscript:n
\char_make_math_subscript:n
\char_make_ignore:n
\char_make_space:n
\char_make_letter:n

```

```

2547 \cs_new_protected_nopar:Npn \char_make_escape:n      #1 { \char_set_catcode:nn {#1} {\c_
2548 \cs_new_protected_nopar:Npn \char_make_begin_group:n #1 { \char_set_catcode:nn {#1} {\c_
2549 \cs_new_protected_nopar:Npn \char_make_end_group:n   #1 { \char_set_catcode:nn {#1} {\c_
2550 \cs_new_protected_nopar:Npn \char_make_math_shift:n  #1 { \char_set_catcode:nn {#1} {\c_
2551 \cs_new_protected_nopar:Npn \char_make_alignment:n   #1 { \char_set_catcode:nn {#1} {\c_
2552 \cs_new_protected_nopar:Npn \char_make_end_line:n    #1 { \char_set_catcode:nn {#1} {\c_
2553 \cs_new_protected_nopar:Npn \char_make_parameter:n   #1 { \char_set_catcode:nn {#1} {\c_
2554 \cs_new_protected_nopar:Npn \char_make_math_superscript:n #1 { \char_set_catcode:nn {#1} {\c_
2555 \cs_new_protected_nopar:Npn \char_make_math_subscript:n #1 { \char_set_catcode:nn {#1} {\c_
2556 \cs_new_protected_nopar:Npn \char_make_ignore:n      #1 { \char_set_catcode:nn {#1} {\c_
2557 \cs_new_protected_nopar:Npn \char_make_space:n       #1 { \char_set_catcode:nn {#1} {\c_
2558 \cs_new_protected_nopar:Npn \char_make_letter:n     #1 { \char_set_catcode:nn {#1} {\c_
2559 \cs_new_protected_nopar:Npn \char_make_other:n       #1 { \char_set_catcode:nn {#1} {\c_
2560 \cs_new_protected_nopar:Npn \char_make_active:n      #1 { \char_set_catcode:nn {#1} {\c_
2561 \cs_new_protected_nopar:Npn \char_make_comment:n     #1 { \char_set_catcode:nn {#1} {\c_
2562 \cs_new_protected_nopar:Npn \char_make_invalid:n     #1 { \char_set_catcode:nn {#1} {\c_

```

(End definition for `\char_make_escape:n` and others. These functions are documented on page 47.)

Math codes.

```

\char_set_mathcode:w \char_set_mathcode:nn \char_gset_mathcode:w \char_gset_mathcode:nn
\char_value_mathcode:w \char_value_mathcode:n \char_show_value_mathcode:w \char_show_value_mathcode:n
2563 \cs_new_eq:NN \char_set_mathcode:w \tex_mathcode:D
2564 \cs_new_protected_nopar:Npn \char_set_mathcode:nn #1#2 {
2565   \char_set_mathcode:w #1 = \int_eval:w #2\int_eval_end:
2566 }
2567 \cs_new_protected_nopar:Npn \char_gset_mathcode:w { \pref_global:D \tex_mathcode:D }
2568 \cs_new_protected_nopar:Npn \char_gset_mathcode:nn #1#2 {
2569   \char_gset_mathcode:w #1 = \int_eval:w #2\int_eval_end:
2570 }
2571 \cs_new_nopar:Npn \char_value_mathcode:w { \int_use:N \tex_mathcode:D }
2572 \cs_new_nopar:Npn \char_value_mathcode:n #1 {
2573   \char_value_mathcode:w \int_eval:w #1\int_eval_end:
2574 }
2575 \cs_new_nopar:Npn \char_show_value_mathcode:w { \tex_showthe:D \tex_mathcode:D }
2576 \cs_new_nopar:Npn \char_show_value_mathcode:n #1 {
2577   \char_show_value_mathcode:w \int_eval:w #1\int_eval_end:
2578 }

```

(End definition for `\char_set_mathcode:w` and others. These functions are documented on page 49.)

```

\char_set_lccode:w \char_set_lccode:nn \char_value_lccode:w \char_value_lccode:n
\char_show_value_lccode:w \char_show_value_lccode:n
2579 \cs_new_eq:NN \char_set_lccode:w \tex_lccode:D
2580 \cs_new_protected_nopar:Npn \char_set_lccode:nn #1#2{
2581   \char_set_lccode:w #1 = \int_eval:w #2\int_eval_end:
2582 }
2583 \cs_new_nopar:Npn \char_value_lccode:w {\int_use:N\tex_lccode:D}
2584 \cs_new_nopar:Npn \char_value_lccode:n #1{\char_value_lccode:w
2585   \int_eval:w #1\int_eval_end:}
2586 \cs_new_nopar:Npn \char_show_value_lccode:w {\tex_showthe:D\tex_lccode:D}

```



```

2587 \cs_new_nopar:Npn \char_show_value_lccode:n #1{
2588   \char_show_value_lccode:w \int_eval:w #1\int_eval_end:}

```

(End definition for `\char_set_lccode:w` and others. These functions are documented on page 48.)

```

\char_set_uccode:w
\char_set_uccode:nn
\char_value_uccode:w
\char_value_uccode:n
\char_show_value_uccode:w
\char_show_value_uccode:n

2589 \cs_new_eq:NN \char_set_uccode:w \tex_uccode:D
2590 \cs_new_protected_nopar:Npn \char_set_uccode:nn #1#2{
2591   \char_set_uccode:w #1 = \int_eval:w #2\int_eval_end:
2592 }
2593 \cs_new_nopar:Npn \char_value_uccode:w {\int_use:N\tex_uccode:D}
2594 \cs_new_nopar:Npn \char_value_uccode:n #1{\char_value_uccode:w
2595   \int_eval:w #1\int_eval_end:}
2596 \cs_new_nopar:Npn \char_show_value_uccode:w {\tex_showthe:D\tex_uccode:D}
2597 \cs_new_nopar:Npn \char_show_value_uccode:n #1{
2598   \char_show_value_uccode:w \int_eval:w #1\int_eval_end:}

```

(End definition for `\char_set_uccode:w` and others. These functions are documented on page 48.)

```

\char_set_sfcode:w
\char_set_sfcode:nn
\char_value_sfcode:w
\char_value_sfcode:n
\char_show_value_sfcode:w
\char_show_value_sfcode:n

2599 \cs_new_eq:NN \char_set_sfcode:w \tex_sfcode:D
2600 \cs_new_protected_nopar:Npn \char_set_sfcode:nn #1#2 {
2601   \char_set_sfcode:w #1 = \int_eval:w #2\int_eval_end:
2602 }
2603 \cs_new_nopar:Npn \char_value_sfcode:w { \int_use:N \tex_sfcode:D }
2604 \cs_new_nopar:Npn \char_value_sfcode:n #1 {
2605   \char_value_sfcode:w \int_eval:w #1\int_eval_end:
2606 }
2607 \cs_new_nopar:Npn \char_show_value_sfcode:w { \tex_showthe:D \tex_sfcode:D }
2608 \cs_new_nopar:Npn \char_show_value_sfcode:n #1 {
2609   \char_show_value_sfcode:w \int_eval:w #1\int_eval_end:
2610 }

```

(End definition for `\char_set_sfcode:w` and others. These functions are documented on page 48.)

99.4 Generic tokens

`\token_new:Nn` Creates a new token.

```

2611 \cs_new_protected_nopar:Npn \token_new:Nn #1#2 {\cs_new_eq:NN #1#2}

```

(End definition for `\token_new:Nn`. This function is documented on page 49.)

```

\c_group_begin_token
\c_group_end_token
\c_math_shift_token
\c_alignment_tab_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token
\c_letter_token
\c_other_char_token
\c_active_char_token

```

We define these useful tokens. We have to do it by hand with the brace tokens for obvious reasons.

```

2612 \cs_new_eq:NN \c_group_begin_token {
2613 \cs_new_eq:NN \c_group_end_token }

```

```

2614 \group_begin:
2615 \char_set_catcode:nn{'\*}{3}
2616 \token_new:Nn \c_math_shift_token {*}
2617 \char_set_catcode:nn{'\*}{4}
2618 \token_new:Nn \c_alignment_tab_token {*}
2619 \token_new:Nn \c_parameter_token {#}
2620 \token_new:Nn \c_math_superscript_token {^}
2621 \char_set_catcode:nn{'\*}{8}
2622 \token_new:Nn \c_math_subscript_token {*}
2623 \token_new:Nn \c_space_token {~}
2624 \token_new:Nn \c_letter_token {a}
2625 \token_new:Nn \c_other_char_token {1}
2626 \char_set_catcode:nn{'\*}{13}
2627 \cs_gset_nopar:Npn \c_active_char_token {\exp_not:N*}
2628 \group_end:

```

(End definition for `\c_group_begin_token` and others. These functions are documented on page 49.)

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.
`\token_if_group_begin:NTF`

```

2629 \prg_new_conditional:Nnn \token_if_group_begin:N {p,TF,T,F} {
2630   \if_catcode:w \exp_not:N #1\c_group_begin_token
2631   \prg_return_true: \else: \prg_return_false: \fi:
2632 }

```

(End definition for `\token_if_group_begin_p:N`. This function is documented on page 49.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.
`\token_if_group_end:NTF`

```

2633 \prg_new_conditional:Nnn \token_if_group_end:N {p,TF,T,F} {
2634   \if_catcode:w \exp_not:N #1\c_group_end_token
2635   \prg_return_true: \else: \prg_return_false: \fi:
2636 }

```

(End definition for `\token_if_group_end_p:N`. This function is documented on page 49.)

`\token_if_math_shift_p:N` Check if token is a math shift token. We use the constant `\c_math_shift_token` for this.
`\token_if_math_shift:NTF`

```

2637 \prg_new_conditional:Nnn \token_if_math_shift:N {p,TF,T,F} {
2638   \if_catcode:w \exp_not:N #1\c_math_shift_token
2639   \prg_return_true: \else: \prg_return_false: \fi:
2640 }

```

(End definition for `\token_if_math_shift_p:N`. This function is documented on page 50.)

`\token_if_alignment_tab_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_tab_token` for this.
`\token_if_alignment_tab:NTF`

```

2641 \prg_new_conditional:Nnn \token_if_alignment_tab:N {p,TF,T,F} {
2642   \if_catcode:w \exp_not:N #1\c_alignment_tab_token
2643   \prg_return_true: \else: \prg_return_false: \fi:
2644 }

```

(End definition for `\token_if_alignment_tab_p:N`. This function is documented on page 50.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.
`\token_if_parameter:NTF` We have to trick \TeX a bit to avoid an error message.

```

2645 \prg_new_conditional:Nnn \token_if_parameter:N {p,TF,T,F} {
2646   \exp_after:wN\if_catcode:w \cs:w c_parameter_token\cs_end:\exp_not:N #1
2647   \prg_return_true: \else: \prg_return_false: \fi:
2648 }

```

(End definition for `\token_if_parameter_p:N`. This function is documented on page 50.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token`
`\token_if_math_superscript:NTF` for this.

```

2649 \prg_new_conditional:Nnn \token_if_math_superscript:N {p,TF,T,F} {
2650   \if_catcode:w \exp_not:N #1\c_math_superscript_token
2651   \prg_return_true: \else: \prg_return_false: \fi:
2652 }

```

(End definition for `\token_if_math_superscript_p:N`. This function is documented on page 50.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token`
`\token_if_math_subscript:NTF` for this.

```

2653 \prg_new_conditional:Nnn \token_if_math_subscript:N {p,TF,T,F} {
2654   \if_catcode:w \exp_not:N #1\c_math_subscript_token
2655   \prg_return_true: \else: \prg_return_false: \fi:
2656 }

```

(End definition for `\token_if_math_subscript_p:N`. This function is documented on page 50.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.
`\token_if_space:NTF`

```

2657 \prg_new_conditional:Nnn \token_if_space:N {p,TF,T,F} {
2658   \if_catcode:w \exp_not:N #1\c_space_token
2659   \prg_return_true: \else: \prg_return_false: \fi:
2660 }

```

(End definition for `\token_if_space_p:N`. This function is documented on page 50.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_letter_token` for this.
`\token_if_letter:NTF`

```

2661 \prg_new_conditional:Nnn \token_if_letter:N {p,TF,T,F} {
2662   \if_catcode:w \exp_not:N #1\c_letter_token
2663   \prg_return_true: \else: \prg_return_false: \fi:
2664 }

```

(End definition for `\token_if_letter_p:N`. This function is documented on page 50.)

`\token_if_other_char_p:N` Check if token is an other char token. We use the constant `\c_other_char_token` for
`\token_if_other_char:NTF` this.

```
2665 \prg_new_conditional:Nnn \token_if_other_char:N {p,TF,T,F} {  
2666   \if_catcode:w \exp_not:N #1\c_other_char_token  
2667   \prg_return_true: \else: \prg_return_false: \fi:  
2668 }
```

(End definition for `\token_if_other_char_p:N`. This function is documented on page 50.)

`\token_if_active_char_p:N` Check if token is an active char token. We use the constant `\c_active_char_token` for
`\token_if_active_char:NTF` this.

```
2669 \prg_new_conditional:Nnn \token_if_active_char:N {p,TF,T,F} {  
2670   \if_catcode:w \exp_not:N #1\c_active_char_token  
2671   \prg_return_true: \else: \prg_return_false: \fi:  
2672 }
```

(End definition for `\token_if_active_char_p:N`. This function is documented on page 51.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.
`\token_if_eq_meaning:NNTF`

```
2673 \prg_new_conditional:Nnn \token_if_eq_meaning:NN {p,TF,T,F} {  
2674   \if_meaning:w #1 #2  
2675   \prg_return_true: \else: \prg_return_false: \fi:  
2676 }
```

(End definition for `\token_if_eq_meaning_p:NN`. This function is documented on page 51.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.
`\token_if_eq_catcode:NNTF`

```
2677 \prg_new_conditional:Nnn \token_if_eq_catcode:NN {p,TF,T,F} {  
2678   \if_catcode:w \exp_not:N #1 \exp_not:N #2  
2679   \prg_return_true: \else: \prg_return_false: \fi:  
2680 }
```

(End definition for `\token_if_eq_catcode_p:NN`. This function is documented on page 51.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.
`\token_if_eq_charcode:NNTF`

```
2681 \prg_new_conditional:Nnn \token_if_eq_charcode:NN {p,TF,T,F} {  
2682   \if_charcode:w \exp_not:N #1 \exp_not:N #2  
2683   \prg_return_true: \else: \prg_return_false: \fi:  
2684 }
```

(End definition for `\token_if_eq_charcode_p:NN`. This function is documented on page 51.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` will always output something like
`\token_if_macro:NTF` `\long macro:#1->#1` so we simply check to see if the meaning contains `->`. Argument
`\token_if_macro_p_aux:w` `#2` in the code below will be empty if the string `->` isn't present, proof that the token was
not a macro (which is why we reverse the emptiness test). However this function will fail
on its own auxiliary function (and a few other private functions as well) but that should
certainly never be a problem!

```

2685 \prg_new_conditional:Nnn \token_if_macro:N {p,TF,T,F} {
2686   \exp_after:wN \token_if_macro_p_aux:w \token_to_meaning:N #1 -> \q_stop
2687 }
2688 \cs_new_nopar:Npn \token_if_macro_p_aux:w #1 -> #2 \q_stop{
2689   \if_predicate:w \tl_if_empty_p:n{#2}
2690     \prg_return_false: \else: \prg_return_true: \fi:
2691 }

```

(End definition for `\token_if_macro_p:N`. This function is documented on page 51.)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. We use `\scan_stop:` for this.
`\token_if_cs:NTF`

```

2692 \prg_new_conditional:Nnn \token_if_cs:N {p,TF,T,F} {
2693   \if_predicate:w \token_if_eq_catcode_p:NN \scan_stop: #1
2694     \prg_return_true: \else: \prg_return_false: \fi:

```

(End definition for `\token_if_cs_p:N`. This function is documented on page 51.)

`\token_if_expandable_p:N` Check if token is expandable. We use the fact that T_EX will temporarily convert
`\token_if_expandable:NTF` `\exp_not:N {token}` into `\scan_stop:` if `{token}` is expandable.

```

2695 \prg_new_conditional:Nnn \token_if_expandable:N {p,TF,T,F} {
2696   \cs_if_exist:NTF #1 {
2697     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2698     \prg_return_false: \else: \prg_return_true: \fi:
2699   } {
2700     \prg_return_false:
2701   }
2702 }

```

(End definition for `\token_if_expandable_p:N`. This function is documented on page 51.)

`\token_if_chardef_p:N` Most of these functions have to check the meaning of the token in question so we need to
`\token_if_mathchardef_p:N` do some checkups on which characters are output by `\token_to_meaning:N`. As usual,
`\token_if_int_register_p:N` these characters have catcode 12 so we must do some serious substitutions in the code
`\token_if_skip_register_p:N` below...
`\token_if_dim_register_p:N`

```

2703 \group_begin:
2704   \char_set_lccode:nn {'\T}{'\T}
2705   \char_set_lccode:nn {'\F}{'\F}
2706   \char_set_lccode:nn {'\X}{'\n}
2707   \char_set_lccode:nn {'\Y}{'\t}
2708   \char_set_lccode:nn {'\Z}{'\d}

```

```

\token_if_protected_macro:NTF
  \token_if_protected_long_macro:NTF
\token_if_dim_register:NTF
\token_if_skip_register:NTF
\token_if_int_register:NTF
\token_if_toks_register:NTF
\token_if_chardef_p_aux:w
  \token_if_mathchardef_p_aux:w

```

```

2709 \char_set_lccode:nn {'\?}{'\}
2710 \tl_map_inline:nn{\X\Y\Z\M\C\H\A\R\O\U\S\K\I\P\L\G\P\E}
2711   {\char_set_catcode:nn {'#1}{12}}

```

We convert the token list to lowercase and restore the catcode and lowercase code changes.

```

2712 \tl_to_lowercase:n{
2713   \group_end:

```

First up is checking if something has been defined with `\tex_chardef:D` or `\tex_mathchardef:D`. This is easy since \TeX thinks of such tokens as hexadecimal so it stores them as `\char"<hex number>` or `\mathchar"<hex number>`.

```

2721 \prg_new_conditional:Nnn \token_if_chardef:N {p,TF,T,F} {
2722   \exp_after:wN \token_if_chardef_aux:w
2723   \token_to_meaning:N #1?CHAR"\q_stop
2724 }
2725 \cs_new_nopar:Npn \token_if_chardef_aux:w #1?CHAR"#2\q_stop{
2726   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2727 }

2728 \prg_new_conditional:Nnn \token_if_mathchardef:N {p,TF,T,F} {
2729   \exp_after:wN \token_if_mathchardef_aux:w
2730   \token_to_meaning:N #1?MAYHCHAR"\q_stop
2731 }
2732 \cs_new_nopar:Npn \token_if_mathchardef_aux:w #1?MAYHCHAR"#2\q_stop{
2733   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2734 }

```

Integer registers are a little more difficult since they expand to `\count<number>` and there is also a primitive `\countdef`. So we have to check for that primitive as well.

```

2735 \prg_new_conditional:Nnn \token_if_int_register:N {p,TF,T,F} {
2736   \if_meaning:w \tex_countdef:D #1
2737   \prg_return_false:
2738 }
2739 \else:
2740   \exp_after:wN \token_if_int_register_aux:w
2741   \token_to_meaning:N #1?COUXY\q_stop
2742   \fi:
2743 }
2744 \cs_new_nopar:Npn \token_if_int_register_aux:w #1?COUXY#2\q_stop{
2745   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2746 }

```

Skip registers are done the same way as the integer registers.

```

2747 \prg_new_conditional:Nnn \token_if_skip_register:N {p,TF,T,F} {
2748   \if_meaning:w \tex_skipdef:D #1
2749   \prg_return_false:
2750 }
2751 \else:
2752   \exp_after:wN \token_if_skip_register_aux:w

```

```

2744     \token_to_meaning:N #1?SKIP\q_stop
2745     \fi:
2746   }
2747   \cs_new_nopar:Npn \token_if_skip_register_aux:w #1?SKIP#2\q_stop{
2748     \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2749   }

```

Dim registers. No news here

```

2750   \prg_new_conditional:Nnn \token_if_dim_register:N {p,TF,T,F} {
2751     \if_meaning:w \tex_dimendef:D #1
2752     \c_false_bool
2753   \else:
2754     \exp_after:wN \token_if_dim_register_aux:w
2755     \token_to_meaning:N #1?ZIMEX\q_stop
2756   \fi:
2757 }
2758 \cs_new_nopar:Npn \token_if_dim_register_aux:w #1?ZIMEX#2\q_stop{
2759   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2760 }

```

Toks registers.

```

2761   \prg_new_conditional:Nnn \token_if_toks_register:N {p,TF,T,F} {
2762     \if_meaning:w \tex_toksdef:D #1
2763     \prg_return_false:
2764   \else:
2765     \exp_after:wN \token_if_toks_register_aux:w
2766     \token_to_meaning:N #1?YOKS\q_stop
2767   \fi:
2768 }
2769 \cs_new_nopar:Npn \token_if_toks_register_aux:w #1?YOKS#2\q_stop{
2770   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2771 }

```

Protected macros.

```

2772   \prg_new_conditional:Nnn \token_if_protected_macro:N {p,TF,T,F} {
2773     \exp_after:wN \token_if_protected_macro_aux:w
2774     \token_to_meaning:N #1?PROYECY EZ~MACRO\q_stop
2775   }
2776   \cs_new_nopar:Npn \token_if_protected_macro_aux:w #1?PROYECY EZ~MACRO#2\q_stop{
2777     \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2778   }

```

Long macros.

```

2779   \prg_new_conditional:Nnn \token_if_long_macro:N {p,TF,T,F} {
2780     \exp_after:wN \token_if_long_macro_aux:w
2781     \token_to_meaning:N #1?LOXG~MACRO\q_stop
2782   }

```

```

2783 \cs_new_nopar:Npn \token_if_long_macro_aux:w #1?LOXG-MACRO#2\q_stop{
2784   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2785 }

```

Finally protected long macros where we for once don't have to add an extra test since there is no primitive for the combined prefixes.

```

2786 \prg_new_conditional:Nnn \token_if_protected_long_macro:N {p,TF,T,F} {
2787   \exp_after:wN \token_if_protected_long_macro_aux:w
2788   \token_to_meaning:N #1?PROYECY EZ?LOXG-MACRO\q_stop
2789 }
2790 \cs_new_nopar:Npn \token_if_protected_long_macro_aux:w #1
2791   ?PROYECY EZ?LOXG-MACRO#2\q_stop{
2792   \tl_if_empty:nTF {#1} {\prg_return_true:} {\prg_return_false:}
2793 }

```

Finally the `\tl_to_lowercase:n` ends!

```

2794 }

```

(End definition for `\token_if_chardef_p:N` and others. These functions are documented on page 53.)

We do not provide a function for testing if a control sequence is “outer” since we don't use that in L^AT_EX3.

```

\token_get_prefix_arg_replacement_aux:w
\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N

```

In the `xparse` package we sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the argument specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

```

2795 \group_begin:
2796 \char_set_lccode:nn {'\'}{'\':}
2797 \char_set_catcode:nn{'M'}{12}
2798 \char_set_catcode:nn{'A'}{12}
2799 \char_set_catcode:nn{'C'}{12}
2800 \char_set_catcode:nn{'R'}{12}
2801 \char_set_catcode:nn{'O'}{12}
2802 \tl_to_lowercase:n{
2803   \group_end:
2804   \cs_new_nopar:Npn \token_get_prefix_arg_replacement_aux:w #1MACRO?#2->#3\q_stop#4{
2805     #4{#1}{#2}{#3}
2806   }
2807   \cs_new_nopar:Npn \token_get_prefix_spec:N #1{
2808     \token_if_macro:NTF #1{
2809       \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2810       \token_to_meaning:N #1\q_stop\use_i:nnn
2811     }{\scan_stop:}
2812   }

```



```

2813 \cs_new_nopar:Npn\token_get_arg_spec:N #1{
2814   \token_if_macro:NTF #1{
2815     \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2816     \token_to_meaning:N #1\q_stop\use_ii:nnn
2817   }{\scan_stop:}
2818 }
2819 \cs_new_nopar:Npn\token_get_replacement_spec:N #1{
2820   \token_if_macro:NTF #1{
2821     \exp_after:wN \token_get_prefix_arg_replacement_aux:w
2822     \token_to_meaning:N #1\q_stop\use_iii:nnn
2823   }{\scan_stop:}
2824 }
2825 }

```

(End definition for \token_get_prefix_arg_replacement_aux:w.)

Useless code: because we can!

```

\token_if_primitive_p:N
\token_if_primitive_p_aux:N
\token_if_primitive:NTF

```

It is rather hard to determine if a token is a primitive. First we can check if it is a control sequence or active character. If either, we check if it is a macro. Then we can go through a tedious process of testing for different register types... I don't actually think this function is useful but you never know.

```

2826 \prg_new_conditional:Nnn \token_if_primitive:N {p,TF,T,F} {
2827   \if_predicate:w \token_if_cs_p:N #1
2828   \if_predicate:w \token_if_macro_p:N #1
2829   \prg_return_false:
2830   \else:
2831     \token_if_primitive_p_aux:N #1
2832   \fi:
2833   \else:
2834     \if_predicate:w \token_if_active_char_p:N #1
2835     \if_predicate:w \token_if_macro_p:N #1
2836     \prg_return_false:
2837     \else:
2838       \token_if_primitive_p_aux:N #1
2839     \fi:
2840     \else:
2841       \prg_return_false:
2842     \fi:
2843   \fi:
2844 }
2845 \cs_new_nopar:Npn \token_if_primitive_p_aux:N #1{
2846   \if_predicate:w \token_if_chardef_p:N #1 \c_false_bool
2847   \else:
2848     \if_predicate:w \token_if_mathchardef_p:N #1 \prg_return_false:
2849     \else:
2850       \if_predicate:w \token_if_int_register_p:N #1 \prg_return_false:
2851       \else:

```

```

2852         \if_predicate:w \token_if_skip_register_p:N #1 \prg_return_false:
2853     \else:
2854         \if_predicate:w \token_if_dim_register_p:N #1 \prg_return_false:
2855     \else:
2856         \if_predicate:w \token_if_toks_register_p:N #1 \prg_return_false:
2857     \else:

```

We made it!

```

2858         \prg_return_true:
2859     \fi:
2860 \fi:
2861 \fi:
2862 \fi:
2863 \fi:
2864 \fi:
2865 }

```

(End definition for `\token_if_primitive_p:N`. This function is documented on page 53.)

99.5 Peeking ahead at the next token

`\l_peek_token`
`\g_peek_token`
`\l_peek_search_token`

We define some other tokens which will initially be the character ?.

```

2866 \token_new:Nn \l_peek_token {?}
2867 \token_new:Nn \g_peek_token {?}
2868 \token_new:Nn \l_peek_search_token {?}

```

(End definition for `\l_peek_token`. This function is documented on page 53.)

`\peek_after:NN`
`\peek_gafter:NN`

`\peek_after:NN` takes two argument where the first is a function acting on `\l_peek_token` and the second is the next token in the input stream which `\l_peek_token` is set equal to. `\peek_gafter:NN` does the same globally to `\g_peek_token`.

```

2869 \cs_new_protected_nopar:Npn \peek_after:NN {\tex_futurelet:D \l_peek_token }
2870 \cs_new_protected_nopar:Npn \peek_gafter:NN {
2871     \pref_global:D \tex_futurelet:D \g_peek_token
2872 }

```

(End definition for `\peek_after:NN`. This function is documented on page 53.)

For normal purposes there are four main cases:

1. peek at the next token.
2. peek at the next non-space token.
3. peek at the next token and remove it.
4. peek at the next non-space token and remove it.

The generic functions will take four arguments: The token to search for, the test function to run on it and the true/false cases. The general algorithm is this:

1. Store the token to search for in `\l_peek_search_token`.
2. In order to avoid doubling of hash marks where it seems unnatural we put the $\langle true \rangle$ and $\langle false \rangle$ cases through an `x` type expansion but using `\exp_not:n` to avoid any expansion. This has the same effect as putting it through a $\langle toks \rangle$ register but is faster. Also put in a special alignment safe group end.
3. Put in an alignment safe group begin.
4. Peek ahead and call the function which will act on the next token in the input stream.

`\l_peek_true_tl` Two dedicated token list variables that store the true and false cases.
`\l_peek_false_tl`

```
2873 \tl_new:N \l_peek_true_tl
2874 \tl_new:N \l_peek_false_tl
```

(End definition for `\l_peek_true_tl`. This function is documented on page 247.)

`\peek_tmp:w` Scratch function used for storing the token to be removed if found.

```
2875 \cs_new_nopar:Npn \peek_tmp:w {}
```

(End definition for `\peek_tmp:w`. This function is documented on page 247.)

`\l_peek_search_tl` We also use this token list variable for storing the token we want to compare. This turns out to be useful.

```
2876 \tl_new:N \l_peek_search_tl
```

(End definition for `\l_peek_search_tl`. This function is documented on page 247.)

`\peek_token_generic:NNTF` #1 : the function to execute (obey or ignore spaces, etc.),
#2 : the special token we're looking for.

```
2877 \cs_new_protected:Npn \peek_token_generic:NNTF #1#2#3#4 {
2878   \cs_set_eq:NN \l_peek_search_token #2
2879   \tl_set:Nn \l_peek_search_tl {#2}
2880   \tl_set:Nn \l_peek_true_tl { \group_align_safe_end: #3 }
2881   \tl_set:Nn \l_peek_false_tl { \group_align_safe_end: #4 }
2882   \group_align_safe_begin:
2883     \peek_after:NN #1
2884 }
2885 \cs_new_protected:Npn \peek_token_generic:NNT #1#2#3 {
2886   \peek_token_generic:NNTF #1#2 {#3} {}
2887 }
2888 \cs_new_protected:Npn \peek_token_generic:NNTF #1#2#3 {
2889   \peek_token_generic:NNTF #1#2 {} {#3}
2890 }
```

(End definition for `\peek_token_remove_generic:NN`. This function is documented on page 54.)

`\peek_token_remove_generic:NNTF` If we want to be able to remove any character from the input stream we might as well do it the same way for all characters so we define this as little differently from above.

```

2891 \cs_new_protected:Npn \peek_token_remove_generic:NNTF #1#2#3#4 {
2892   \cs_set_eq:NN \l_peek_search_token #2
2893   \tl_set:Nn \l_peek_search_tl {#2}
2894   \tl_set:Nn \l_peek_true_aux_tl {#3}
2895   \tl_set_eq:NN \l_peek_true_tl \c_peek_true_remove_next_tl
2896   \tl_set:Nn \l_peek_false_tl {\group_align_safe_end: #4}
2897   \group_align_safe_begin:
2898     \peek_after:NN #1
2899   }
2900 \cs_new:Npn \peek_token_remove_generic:NNT #1#2#3 {
2901   \peek_token_remove_generic:NNTF #1#2 {#3} {}
2902 }
2903 \cs_new:Npn \peek_token_remove_generic:NNF #1#2#3 {
2904   \peek_token_remove_generic:NNTF #1#2 {} {#3}
2905 }

```

(End definition for `\peek_token_remove_generic:NN`. This function is documented on page 54.)

`\l_peek_true_aux_tl` `\c_peek_true_remove_next_tl` Two token list variables to help with removing the character from the input stream.

```

2906 \tl_new:N \l_peek_true_aux_tl
2907 \tl_const:Nn \c_peek_true_remove_next_tl {\group_align_safe_end:
2908   \tex_afterassignment:D \l_peek_true_aux_tl \cs_set_eq:NN \peek_tmp:w
2909 }

```

(End definition for `\l_peek_true_aux_tl`. This function is documented on page 247.)

`\peek_execute_branches_meaning:` `\peek_execute_branches_catcode:` `\peek_execute_branches_charcode:` There are three major tests between tokens in T_EX: meaning, catcode and charcode. Hence we define three basic test functions that set in after the ignoring phase is over and done with.

```

\peek_execute_branches_charcode_aux:NN
2910 \cs_new_nopar:Npn \peek_execute_branches_meaning: {
2911   \if_meaning:w \l_peek_token \l_peek_search_token
2912     \exp_after:wN \l_peek_true_tl
2913   \else:
2914     \exp_after:wN \l_peek_false_tl
2915   \fi:
2916 }
2917 \cs_new_nopar:Npn \peek_execute_branches_catcode: {
2918   \if_catcode:w \exp_not:N\l_peek_token \exp_not:N\l_peek_search_token
2919     \exp_after:wN \l_peek_true_tl
2920   \else:
2921     \exp_after:wN \l_peek_false_tl
2922   \fi:
2923 }

```

For the charcode version we do things a little differently. We want to check the token directly but if we do this we face problems if the next thing in the input stream is a braced group or a space token. The braced group would be read as a complete argument and the space would be gobbled by T_EX's argument reading routines. Hence we test for both of these and if one of them is found we just execute the false result directly since no one should ever try to use the `charcode` function for searching for `\c_group_begin_token` or `\c_space_token`. The same is true for `\c_group_end_token`, as this can only occur if the function is at the end of a group.

```

2924 \cs_new_nopar:Npn \peek_execute_branches_charcode: {
2925   \bool_if:nTF {
2926     \token_if_eq_catcode_p:NN \l_peek_token \c_group_begin_token ||
2927     \token_if_eq_catcode_p:NN \l_peek_token \c_group_end_token ||
2928     \token_if_eq_meaning_p:NN \l_peek_token \c_space_token
2929   }
2930   { \l_peek_false_tl }

```

Otherwise we call a small auxiliary function that just grabs the next token. We can do that because it really is a single token; we just have insert it again afterwards. Also we stored the token we were looking for in the token list variable `\l_peek_search_tl` so we unpack it again for this function.

```

2931   { \exp_after:wN \peek_execute_branches_charcode_aux:NN \l_peek_search_tl }
2932 }

```

Then we just do the usual `\if_charcode:w` comparison. We also remember to insert `#2` again after executing the true or false branches.

```

2933 \cs_new:Npn \peek_execute_branches_charcode_aux:NN #1#2{
2934   \if_charcode:w \exp_not:N #1\exp_not:N#2
2935     \exp_after:wN \l_peek_true_tl
2936   \else:
2937     \exp_after:wN \l_peek_false_tl
2938   \fi:
2939   #2
2940 }

```

(End definition for `\peek_execute_branches_meaning:.` This function is documented on page 55.)

`\peek_def_aux:nnnn`
`\peek_def_aux_ii:nnnnn`

This function aids defining conditional variants without too much repeated code. I hope that it doesn't detract too much from the readability.

```

2941 \cs_new_nopar:Npn \peek_def_aux:nnnn #1#2#3#4 {
2942   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { TF }
2943   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { T }
2944   \peek_def_aux_ii:nnnnn {#1} {#2} {#3} {#4} { F }
2945 }
2946 \cs_new_protected_nopar:Npn \peek_def_aux_ii:nnnnn #1#2#3#4#5 {
2947   \cs_new_nopar:cpx { #1 #5 } {
2948     \tl_if_empty:nF {#2} {

```

```

2949     \exp_not:n { \cs_set_eq:NN \peek_execute_branches: #2 }
2950   }
2951   \exp_not:c { #3 #5 }
2952   \exp_not:n { #4 }
2953 }
2954 }

```

(End definition for \peek_def_aux:nnnn and \peek_def_aux_ii:nnnn.)

\peek_meaning:NTF Here we use meaning comparison with \if_meaning:w.

```

2955 \peek_def_aux:nnnn
2956 { peek_meaning:N }
2957 {}
2958 { peek_token_generic:NN }
2959 { \peek_execute_branches_meaning: }

```

(End definition for \peek_meaning:N. This function is documented on page 54.)

\peek_meaning_ignore_spaces:NTF

```

2960 \peek_def_aux:nnnn
2961 { peek_meaning_ignore_spaces:N }
2962 { \peek_execute_branches_meaning: }
2963 { peek_token_generic:NN }
2964 { \peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_meaning_ignore_spaces:N. This function is documented on page 54.)

\peek_meaning_remove:NTF

```

2965 \peek_def_aux:nnnn
2966 { peek_meaning_remove:N }
2967 {}
2968 { peek_token_remove_generic:NN }
2969 { \peek_execute_branches_meaning: }

```

(End definition for \peek_meaning_remove:N. This function is documented on page 54.)

\peek_meaning_remove_ignore_spaces:NTF

```

2970 \peek_def_aux:nnnn
2971 { peek_meaning_remove_ignore_spaces:N }
2972 { \peek_execute_branches_meaning: }
2973 { peek_token_remove_generic:NN }
2974 { \peek_ignore_spaces_execute_branches: }

```

(End definition for \peek_meaning_remove_ignore_spaces:N. This function is documented on page 54.)

\peek_catcode:NTF Here we use catcode comparison with \if_catcode:w.

```
2975 \peek_def_aux:nnnn
2976 { peek_catcode:N }
2977 {}
2978 { peek_token_generic:NN }
2979 { \peek_execute_branches_catcode: }
```

(End definition for \peek_catcode:N. This function is documented on page 54.)

\peek_catcode_ignore_spaces:NTF

```
2980 \peek_def_aux:nnnn
2981 { peek_catcode_ignore_spaces:N }
2982 { \peek_execute_branches_catcode: }
2983 { peek_token_generic:NN }
2984 { \peek_ignore_spaces_execute_branches: }
```

(End definition for \peek_catcode_ignore_spaces:N. This function is documented on page 54.)

\peek_catcode_remove:NTF

```
2985 \peek_def_aux:nnnn
2986 { peek_catcode_remove:N }
2987 {}
2988 { peek_token_remove_generic:NN }
2989 { \peek_execute_branches_catcode: }
```

(End definition for \peek_catcode_remove:N. This function is documented on page 54.)

\peek_catcode_remove_ignore_spaces:NTF

```
2990 \peek_def_aux:nnnn
2991 { peek_catcode_remove_ignore_spaces:N }
2992 { \peek_execute_branches_catcode: }
2993 { peek_token_remove_generic:NN }
2994 { \peek_ignore_spaces_execute_branches: }
```

(End definition for \peek_catcode_remove_ignore_spaces:N. This function is documented on page 54.)

\peek_charcode:NTF Here we use charcode comparison with \if_charcode:w.

```
2995 \peek_def_aux:nnnn
2996 { peek_charcode:N }
2997 {}
2998 { peek_token_generic:NN }
2999 { \peek_execute_branches_charcode: }
```

(End definition for \peek_charcode:N. This function is documented on page 54.)

`\peek_charcode_ignore_spaces:NTF`

```
3000 \peek_def_aux:nnnn
3001 { peek_charcode_ignore_spaces:N }
3002 { \peek_execute_branches_charcode: }
3003 { peek_token_generic:NN }
3004 { \peek_ignore_spaces_execute_branches: }
```

(End definition for `\peek_charcode_ignore_spaces:N`. This function is documented on page 54.)

`\peek_charcode_remove:NTF`

```
3005 \peek_def_aux:nnnn
3006 { peek_charcode_remove:N }
3007 {}
3008 { peek_token_remove_generic:NN }
3009 { \peek_execute_branches_charcode: }
```

(End definition for `\peek_charcode_remove:N`. This function is documented on page 54.)

`\peek_charcode_remove_ignore_spaces:NTF`

```
3010 \peek_def_aux:nnnn
3011 { peek_charcode_remove_ignore_spaces:N }
3012 { \peek_execute_branches_charcode: }
3013 { peek_token_remove_generic:NN }
3014 { \peek_ignore_spaces_execute_branches: }
```

(End definition for `\peek_charcode_remove_ignore_spaces:N`. This function is documented on page 54.)

`\peek_ignore_spaces_aux:`

`\peek_ignore_spaces_execute_branches:`

Throw away a space token and search again. We could define this in a more devious way where the auxiliary function gobbles the space token but then what do we do if we decide that a certain function should ignore more than one specific token? For example someone might find it interesting to define a `\peek_` function that ignores a's and b's! Or maybe different kinds of “funny spaces”... Therefore I have decided to use this version which uses `\tex_afterassignment:D` to call the auxiliary function after the next token has been removed by `\cs_set_eq:NN`. That way it is easily extensible.

```
3015 \cs_new_nopar:Npn \peek_ignore_spaces_aux: {
3016   \peek_after:NN \peek_ignore_spaces_execute_branches:
3017 }
3018 \cs_new_protected_nopar:Npn \peek_ignore_spaces_execute_branches: {
3019   \token_if_eq_meaning:NNTF \l_peek_token \c_space_token
3020   { \tex_afterassignment:D \peek_ignore_spaces_aux:
3021     \cs_set_eq:NN \peek_tmp:w
3022   }
3023   \peek_execute_branches:
3024 }
```

(End definition for `\peek_ignore_spaces_aux:` and `\peek_ignore_spaces_execute_branches:`. These functions are documented on page 247.)

```
3025 </initex | package)
```


100 l3int implementation

The following test files are used for this code: *m3int001.lvt,m3int002.lvt,m3int03.lvt*.

100.1 Internal functions and variables

<code>\int_advance:w</code>	<code>\int_advance:w <int register> <optional ‘by’> <number> <space></code>
-----------------------------	---

Increments the count register by the specified amount.

T_EXhackers note: This is T_EX’s `\advance`.

<code>\int_convert_number_to_letter:n *</code>	<code>\int_convert_number_to_letter:n {<integer expression>}</code>
--	---

Internal function for turning a number for a different base into a letter or digit.

<code>\int_pre_eval_one_arg:Nn</code>	<code>\int_pre_eval_one_arg:Nn <function> {<integer expression>}</code>
<code>\int_pre_eval_two_args:Nnn</code>	<code>\int_pre_eval_one_arg:Nnn <function> {<int expr₁>}</code> <code>{<int expr₂>}</code>

These are expansion helpers; they evaluate their integer expressions before handing them off to the specified *<function>*.

<code>\int_get_sign_and_digits:n *</code>	<code>\int_get_sign_and_digits:n {<number>}</code>	
<code>\int_get_sign:n</code>		*
<code>\int_get_digits:n</code>		*

From an argument that may or may not include a + or - sign, these functions expand to the respective components of the number.

100.2 Module loading and primitives definitions

We start by ensuring that the required packages are loaded.

```

3026 <*package>
3027 \ProvidesExplPackage
3028   {\filename}{\filedate}{\fileversion}{\filedescription}
3029 \package_check_loaded_expl:
3030 </package>
3031 <*initex | package>

```

`\int_value:w` Here are the remaining primitives for number comparisons and expressions.

`\int_eval:n`

`\int_eval:w`

`\int_eval_end:`

`\if_int_compare:w`

`\if_int_odd:w`

`\if_num:w`

`\if_case:w`

`\int_to_roman:w`

`\int_advance:w`

```

3032 \cs_set_eq:NN \int_value:w \tex_number:D
3033 \cs_set_eq:NN \int_eval:w \etex_numexpr:D
3034 \cs_set_protected:Npn \int_eval_end: {\tex_relax:D}
3035 \cs_set_eq:NN \if_int_compare:w \tex_ifnum:D
3036 \cs_new_eq:NN \if_num:w \tex_ifnum:D
3037 \cs_set_eq:NN \if_int_odd:w \tex_ifodd:D
3038 \cs_new_eq:NN \if_case:w \tex_ifcase:D
3039 \cs_new_eq:NN \int_to_roman:w \tex_romannumeral:D
3040 \cs_new_eq:NN \int_advance:w \tex_advance:D

```

(End definition for `\int_value:w`. This function is documented on page 266.)

`\int_eval:n` Wrapper for `\int_eval:w`. Can be used in an integer expression or directly in the input stream.

```

3041 \cs_set:Npn \int_eval:n #1{
3042   \int_value:w \int_eval:w #1\int_eval_end:
3043 }

```

(End definition for `\int_eval:n`. This function is documented on page 55.)

100.3 Allocation and setting

`\int_new:N` For the L^AT_EX3 format:

`\int_new:c`

```

3044 \*initex
3045 \alloc_new:nnnN {int} {11} {\c_max_register_int} \tex_countdef:D
3046 \*initex

```

For ‘l3in2e’:

```

3047 \*package
3048 \cs_new_protected_nopar:Npn \int_new:N #1 {
3049   \chk_if_free_cs:N #1
3050   \newcount #1
3051 }
3052 \*package

3053 \cs_generate_variant:Nn \int_new:N {c}

```

(End definition for `\int_new:N` and `\int_new:c`. These functions are documented on page 56.)

`\int_set:Nn` Setting counters is again something that I would like to make uniform at the moment to get a better overview.

`\int_set:cn`

`\int_gset:Nn`

`\int_gset:cn`

```

3054 \cs_new_protected_nopar:Npn \int_set:Nn #1#2{#1 \int_eval:w #2\int_eval_end:
3055 \*check)
3056 \chk_local_or_pref_global:N #1

```

```

3057 </check>
3058 }
3059 \cs_new_protected_nopar:Npn \int_gset:Nn {
3060 <*check>
3061   \pref_global_chk:
3062 </check>
3063 <-check> \pref_global:D
3064   \int_set:Nn }
3065 \cs_generate_variant:Nn \int_set:Nn {cn}
3066 \cs_generate_variant:Nn \int_gset:Nn {cn}

```

(End definition for `\int_set:Nn` and `\int_set:cn`. These functions are documented on page 58.)

`\int_set_eq:NN` Setting equal means using one integer inside the set function of another.

```

\int_set_eq:cn
\int_set_eq:Nc
\int_set_eq:cc
\int_gset_eq:NN
\int_gset_eq:cn
\int_gset_eq:Nc
\int_gset_eq:cc
3067 \cs_new_protected_nopar:Npn \int_set_eq:NN #1#2 {
3068   \int_set:Nn #1 {#2}
3069 }
3070 \cs_generate_variant:Nn \int_set_eq:NN { c }
3071 \cs_generate_variant:Nn \int_set_eq:NN { Nc }
3072 \cs_generate_variant:Nn \int_set_eq:NN { cc }
3073 \cs_new_protected_nopar:Npn \int_gset_eq:NN #1#2 {
3074   \int_gset:Nn #1 {#2}
3075 }
3076 \cs_generate_variant:Nn \int_gset_eq:NN { c }
3077 \cs_generate_variant:Nn \int_gset_eq:NN { Nc }
3078 \cs_generate_variant:Nn \int_gset_eq:NN { cc }

```

(End definition for `\int_set_eq:NN` and others. These functions are documented on page 57.)

`\int_incr:N` Incrementing and decrementing of integer registers is done with the following functions.

```

\int_incr:c
\int_decr:N
\int_decr:c
\int_gincr:N
\int_gincr:c
\int_gdecr:N
\int_gdecr:c
3079 \cs_new_protected_nopar:Npn \int_incr:N #1{\int_advance:w#1\c_one
3080 <*check>
3081   \chk_local_or_pref_global:N #1
3082 </check>
3083 }
3084 \cs_new_protected_nopar:Npn \int_decr:N #1{\int_advance:w#1\c_minus_one
3085 <*check>
3086   \chk_local_or_pref_global:N #1
3087 </check>
3088 }
3089 \cs_new_protected_nopar:Npn \int_gincr:N {

```

We make sure that a local variable is not updated globally by changing the internal test (i.e. `\chk_local_or_pref_global:N`) before making the assignment. This is done by `\pref_global_chk:` which also issues the necessary `\pref_global:D`. This is not very efficient, but this code will be only included for debugging purposes. Using `\pref_global:D` in front of the local function is better in the production versions.

```

3090 <*check>
3091     \pref_global_chk:
3092 </check>
3093 <-check> \pref_global:D
3094     \int_incr:N}
3095 \cs_new_protected_nopar:Npn \int_gdecr:N {
3096 <*check>
3097     \pref_global_chk:
3098 </check>
3099 <-check> \pref_global:D
3100     \int_decr:N}

```

With the `\int_add:Nn` functions we can shorten the above code. If this makes it too slow ...

```

3101 \cs_set_protected_nopar:Npn \int_incr:N #1{\int_add:Nn#1\c_one}
3102 \cs_set_protected_nopar:Npn \int_decr:N #1{\int_add:Nn#1\c_minus_one}
3103 \cs_set_protected_nopar:Npn \int_gincr:N #1{\int_gadd:Nn#1\c_one}
3104 \cs_set_protected_nopar:Npn \int_gdecr:N #1{\int_gadd:Nn#1\c_minus_one}

3105 \cs_generate_variant:Nn \int_incr:N {c}
3106 \cs_generate_variant:Nn \int_decr:N {c}
3107 \cs_generate_variant:Nn \int_gincr:N {c}
3108 \cs_generate_variant:Nn \int_gdecr:N {c}

```

(End definition for `\int_incr:N` and `\int_incr:c`. These functions are documented on page 57.)

`\int_zero:N` Functions that reset an `<int>` register to zero.

```

\int_zero:c
\int_gzero:N
\int_gzero:c
3109 \cs_new_protected_nopar:Npn \int_zero:N #1 {#1=\c_zero}
3110 \cs_generate_variant:Nn \int_zero:N {c}

3111 \cs_new_protected_nopar:Npn \int_gzero:N #1 {\pref_global:D #1=\c_zero}
3112 \cs_generate_variant:Nn \int_gzero:N {c}

```

(End definition for `\int_zero:N` and `\int_zero:c`. These functions are documented on page 58.)

`\int_add:Nn` Adding and subtracting to and from a counter ... We should think of using these functions

```

\int_add:cn
\int_gadd:Nn
\int_gadd:cn
3113 \cs_new_protected_nopar:Npn \int_add:Nn #1#2{

```

`\int_sub:Nn` We need to say by in case the first argument is a register accessed by its number, e.g., `\count23`. Not that it should ever happen but...

```

\int_sub:cn
\int_gsub:Nn
\int_gsub:cn
3114     \int_advance:w #1 by \int_eval:w #2\int_eval_end:
3115 <*check>
3116     \chk_local_or_pref_global:N #1
3117 </check>
3118 }

```

```

3119 \cs_new_nopar:Npn \int_sub:Nn #1#2{
3120     \int_advance:w #1-\int_eval:w #2\int_eval_end:
3121     <*check>
3122     \chk_local_or_pref_global:N #1
3123     </check>
3124 }
3125 \cs_new_protected_nopar:Npn \int_gadd:Nn {
3126     <*check>
3127     \pref_global_chk:
3128     </check>
3129     <-check> \pref_global:D
3130     \int_add:Nn }
3131 \cs_new_protected_nopar:Npn \int_gsub:Nn {
3132     <*check>
3133     \pref_global_chk:
3134     </check>
3135     <-check> \pref_global:D
3136     \int_sub:Nn }
3137 \cs_generate_variant:Nn \int_add:Nn {cn}
3138 \cs_generate_variant:Nn \int_gadd:Nn {cn}
3139 \cs_generate_variant:Nn \int_sub:Nn {cn}
3140 \cs_generate_variant:Nn \int_gsub:Nn {cn}

```

(End definition for `\int_add:Nn` and `\int_add:cn`. These functions are documented on page 58.)

`\int_use:N` Here is how counters are accessed:
`\int_use:c`

```

3141 \cs_new_eq:NN \int_use:N \tex_the:D
3142 \cs_new_nopar:Npn \int_use:c #1{\int_use:N \cs:w#1\cs_end:}

```

(End definition for `\int_use:N` and `\int_use:c`. These functions are documented on page 58.)

`\int_show:N`
`\int_show:c`

```

3143 \cs_new_eq:NN \int_show:N \kernel_register_show:N
3144 \cs_new_eq:NN \int_show:c \kernel_register_show:c

```

(End definition for `\int_show:N` and `\int_show:c`. These functions are documented on page 58.)

`\int_to_arabic:n` Nothing exciting here.

```

3145 \cs_new_nopar:Npn \int_to_arabic:n #1{ \int_eval:n{#1}}

```

(End definition for `\int_to_arabic:n`. This function is documented on page 61.)

`\int_roman_lcuc_mapping:Nnn`

Using T_EX's built-in feature for producing roman numerals has some surprising features. One is the the characters resulting from `\int_to_roman:w` have category code 12 so they may fail in certain comparison tests. Therefore we use a mapping from the character T_EX produces to the character we actually want which will give us letters with category code 11.

```

3146 \cs_new_protected_nopar:Npn \int_roman_lcuc_mapping:Nnn #1#2#3{
3147   \cs_set_nopar:cpn {int_to_lc_roman_#1:}{#2}
3148   \cs_set_nopar:cpn {int_to_uc_roman_#1:}{#3}
3149 }

```

(End definition for `\int_roman_lcuc_mapping:Nnn`.)

Here are the default mappings. I haven't found any examples of say Turkish doing the mapping `i \i I` but at least there is a possibility for it if needed. Note: I have now asked a Turkish person and he tells me they do the `i I` mapping.

```

3150 \int_roman_lcuc_mapping:Nnn i i I
3151 \int_roman_lcuc_mapping:Nnn v v V
3152 \int_roman_lcuc_mapping:Nnn x x X
3153 \int_roman_lcuc_mapping:Nnn l l L
3154 \int_roman_lcuc_mapping:Nnn c c C
3155 \int_roman_lcuc_mapping:Nnn d d D
3156 \int_roman_lcuc_mapping:Nnn m m M

```

For the delimiter we cheat and let it gobble its arguments instead.

```

3157 \int_roman_lcuc_mapping:Nnn Q \use_none:nn \use_none:nn

```

`\int_to_roman:n` The commands for producing the lower and upper case roman numerals run a loop on
`\int_to_Roman:n` one character at a time and also carries some information for upper or lower case with
`\int_to_roman_lcuc:NN` it. We put it through `\int_eval:n` first which is safer and more flexible.

```

3158 \cs_new_nopar:Npn \int_to_roman:n #1 {
3159   \exp_after:wN \int_to_roman_lcuc:NN \exp_after:wN l
3160   \int_to_roman:w \int_eval:n {#1} Q
3161 }
3162 \cs_new_nopar:Npn \int_to_Roman:n #1 {
3163   \exp_after:wN \int_to_roman_lcuc:NN \exp_after:wN u
3164   \int_to_roman:w \int_eval:n {#1} Q
3165 }
3166 \cs_new_nopar:Npn \int_to_roman_lcuc:NN #1#2{
3167   \use:c {int_to_#1c_roman_#2:}
3168   \int_to_roman_lcuc:NN #1
3169 }

```

(End definition for `\int_to_roman:n` and `\int_to_Roman:n`. These functions are documented on page 62.)

`\int_convert_to_symbols:nnn` For conversion of integers to arbitrary symbols the method is in general as follows. The input number (`#1`) is compared to the total number of symbols available at each place (`#2`). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an `f`-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was

small enough then there is no problem and everything is easy. This is more or less the same as `\int_convert_number_with_rule:nnN` but ‘pre-packaged’.

```

3170 \cs_new_nopar:Npn \int_convert_to_symbols:nnn #1#2#3 {
3171   \int_compare:nNnTF {#1} > {#2}
3172   {
3173     \exp_args:Nf \int_convert_to_symbols:nnn
3174     { \int_div_truncate:nn { #1 - 1 } {#2} } {#2} {#3}
3175     \exp_args:Nf \prg_case_int:nnn
3176     { \int_eval:n { 1 + \int_mod:nn { #1 - 1 } {#2} } }
3177     {#3} { }
3178   }
3179   { \exp_args:Nf \prg_case_int:nnn { \int_eval:n {#1} } {#3} { } }
3180 }

```

(End definition for `\int_convert_to_symbols:nnn`. This function is documented on page 63.)

`\int_convert_number_with_rule:nnN` This is our major workhorse for conversions. #1 is the number we want converted, #2 is the base number, and #3 is the function converting the number. This function expects to receive a non-negative integer and as such is ideal for something using `\if_case:w` internally.

The basic example is this: We want to convert the number 50 (#1) into an alphabetic equivalent **ax**. For the English language our list contains 26 elements so this is our argument #2 while the function #3 just turns 1 into a, 2 into b, etc. Hence our goal is to turn 50 into the sequence #3{1}#1{24} so what we do is to first divide 50 by 26 and truncating the result returning 1. Then before we execute this we call the function again but this time on the result of the remainder of the division. This goes on until the remainder is less than or equal to the base number where we just call the function #3 directly on the number.

We do a little pre-expansion of the arguments below as they otherwise have a tendency to grow quite large.

```

3181 \cs_set_nopar:Npn \int_convert_number_with_rule:nnN #1#2#3{
3182   \int_compare:nNnTF {#1}>{#2}
3183   {
3184     \exp_args:Nf \int_convert_number_with_rule:nnN
3185     { \int_div_truncate:nn {#1-1}{#2} }{#2}
3186     #3

```

Note that we have to nudge our modulus function so it won’t return 0 as that wouldn’t work with `\if_case:w` when that expects a positive number to produce a letter.

```

3187     \exp_args:Nf #3 { \int_eval:n{1+\int_mod:nn {#1-1}{#2}} }
3188   }
3189   { \exp_args:Nf #3{ \int_eval:n{#1} } }
3190 }

```

As can be seen it is even simpler to convert to number systems that contain 0, since then we don’t have to add or subtract 1 here and there.

(End definition for `\int_convert_number_with_rule:nnN`. This function is documented on page 67.)

`\int_to_alph:n` These both use the above function with input functions that make sense for the alphabet
`\int_to_Alph:n` in English.

```

3191 \cs_new_nopar:Npn \int_to_alph:n #1 {
3192   \int_convert_to_symbols:nnn {#1} { 26 }
3193   {
3194     { 1 } { a }
3195     { 2 } { b }
3196     { 3 } { c }
3197     { 4 } { d }
3198     { 5 } { e }
3199     { 6 } { f }
3200     { 7 } { g }
3201     { 8 } { h }
3202     { 9 } { i }
3203     { 10 } { j }
3204     { 11 } { k }
3205     { 12 } { l }
3206     { 13 } { m }
3207     { 14 } { n }
3208     { 15 } { o }
3209     { 16 } { p }
3210     { 17 } { q }
3211     { 18 } { r }
3212     { 19 } { s }
3213     { 20 } { t }
3214     { 21 } { u }
3215     { 22 } { v }
3216     { 23 } { w }
3217     { 24 } { x }
3218     { 25 } { y }
3219     { 26 } { z }
3220   }
3221 }
3222 \cs_new_nopar:Npn \int_to_Alph:n #1 {
3223   \int_convert_to_symbols:nnn {#1} { 26 }
3224   {
3225     { 1 } { A }
3226     { 2 } { B }
3227     { 3 } { C }
3228     { 4 } { D }
3229     { 5 } { E }
3230     { 6 } { F }
3231     { 7 } { G }
3232     { 8 } { H }
3233     { 9 } { I }
3234     { 10 } { J }
3235     { 11 } { K }
3236     { 12 } { L }
3237     { 13 } { M }

```



```

3238     { 14 } { N }
3239     { 15 } { O }
3240     { 16 } { P }
3241     { 17 } { Q }
3242     { 18 } { R }
3243     { 19 } { S }
3244     { 20 } { T }
3245     { 21 } { U }
3246     { 22 } { V }
3247     { 23 } { W }
3248     { 24 } { X }
3249     { 25 } { Y }
3250     { 26 } { Z }
3251   }
3252 }

```

(End definition for `\int_to_alph:n` and `\int_to_Alph:n`. These functions are documented on page 61.)

`\int_to_symbol:n` Turning a number into a symbol is also easy enough.

```

3253 \cs_new_nopar:Npn \int_to_symbol:n #1{
3254   \mode_if_math:TF
3255   {
3256     \int_convert_number_with_rule:nnN {#1}{9}
3257     \int_symbol_math_conversion_rule:n
3258   }
3259   {
3260     \int_convert_number_with_rule:nnN {#1}{9}
3261     \int_symbol_text_conversion_rule:n
3262   }
3263 }

```

(End definition for `\int_to_symbol:n`. This function is documented on page 62.)

`\int_symbol_math_conversion_rule:n` Nothing spectacular here.

`\int_symbol_text_conversion_rule:n`

```

3264 \cs_new_nopar:Npn \int_symbol_math_conversion_rule:n #1 {
3265   \if_case:w #1
3266   \or: *
3267   \or: \dagger
3268   \or: \ddagger
3269   \or: \mathsection
3270   \or: \mathparagraph
3271   \or: \!
3272   \or: **
3273   \or: \dagger\dagger
3274   \or: \ddagger\ddagger
3275   \fi:
3276 }
3277 \cs_new_nopar:Npn \int_symbol_text_conversion_rule:n #1 {

```

```

3278 \if_case:w #1
3279 \or: \textasteriskcentered
3280 \or: \textdagger
3281 \or: \textdaggerdbl
3282 \or: \textsection
3283 \or: \textparagraph
3284 \or: \textbardbl
3285 \or: \textasteriskcentered\textasteriskcentered
3286 \or: \textdagger\textdagger
3287 \or: \textdaggerdbl\textdaggerdbl
3288 \fi:
3289 }

```

(End definition for `\int_symbol_math_conversion_rule:n`. This function is documented on page 67.)

```

\l_tmpa_int We provide four local and two global scratch counters, maybe we need more or less.
\l_tmpb_int
\l_tmpc_int 3290 \int_new:N \l_tmpa_int
\g_tmpa_int 3291 \int_new:N \l_tmpb_int
\g_tmpb_int 3292 \int_new:N \l_tmpc_int
3293 \int_new:N \g_tmpa_int
3294 \int_new:N \g_tmpb_int

```

(End definition for `\l_tmpa_int`.)

`\int_pre_eval_one_arg:Nn` These are handy when handing down values to other functions. All they do is evaluate the number in advance.

`\int_pre_eval_two_args:Nnn`

```

3295 \cs_set_nopar:Npn \int_pre_eval_one_arg:Nn #1#2{
3296 \exp_args:Nf#1{\int_eval:n{#2}}
3297 \cs_set_nopar:Npn \int_pre_eval_two_args:Nnn #1#2#3{
3298 \exp_args:Nff#1{\int_eval:n{#2}}{\int_eval:n{#3}}
3299 }

```

(End definition for `\int_pre_eval_one_arg:Nn`. This function is documented on page 266.)

100.4 Scanning and conversion

`\int_from_roman:n` The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by T_EX.

`\int_from_roman_aux:NN`

`\int_from_roman_end:w`

`\int_from_roman_clean_up:w`

```

3300 \cs_new_nopar:Npn \int_from_roman:n #1 {
3301 \tl_if_blank:nF {#1}
3302 {
3303 \tex_expandafter:D \int_from_roman_end:w
3304 \tex_number:D \etex_numexpr:D
3305 \int_from_roman_aux:NN #1 Q \q_stop
3306 }
3307 }

```

```

3308 \cs_new_nopar:Npn \int_from_roman_aux:NN #1#2 {
3309   \str_if_eq:nnTF {#1} { Q }
3310   {#1#2}
3311   {
3312     \str_if_eq:nnTF {#2} { Q }
3313     {
3314       \cs_if_exist:cF { c_int_from_roman_ #1 _int }
3315       { \int_from_roman_clean_up:w }
3316       +
3317       \use:c { c_int_from_roman_ #1 _int }
3318       #2
3319     }
3320     {
3321       \cs_if_exist:cF { c_int_from_roman_ #1 _int }
3322       { \int_from_roman_clean_up:w }
3323       \cs_if_exist:cF { c_int_from_roman_ #2 _int }
3324       { \int_from_roman_clean_up:w }
3325       \int_compare:nNnTF
3326       { \use:c { c_int_from_roman_ #1 _int } }
3327       <
3328       { \use:c { c_int_from_roman_ #2 _int } }
3329       {
3330         + \use:c { c_int_from_roman_ #2 _int }
3331         - \use:c { c_int_from_roman_ #1 _int }
3332         \int_from_roman_aux:NN
3333       }
3334       {
3335         + \use:c { c_int_from_roman_ #1 _int }
3336         \int_from_roman_aux:NN #2
3337       }
3338     }
3339   }
3340 }
3341 \cs_new_nopar:Npn \int_from_roman_end:w #1 Q #2 \q_stop {
3342   \tl_if_empty:nTF {#2} {#1} {#2}
3343 }
3344 \cs_new_nopar:Npn \int_from_roman_clean_up:w #1 Q { + 0 Q -1 }

```

(End definition for `\int_from_roman:n`. This function is documented on page 63.)

`\int_convert_from_base_ten:nn` Converting from base ten (#1) to a second base (#2) starts with a simple sign check. As
`\int_convert_from_base_ten_aux:nnn` the input is base 10 TeX can then do the actual work with the sign itself.
`\int_convert_number_to_letter:n`

```

3345 \cs_new:Npn \int_convert_from_base_ten:nn #1#2 {
3346   \int_compare:nNnTF {#1} < { 0 }
3347   {
3348     -
3349     \exp_args:Nnf \int_convert_from_base_ten_aux:nnn
3350     { } { \int_eval:n { 0 - ( #1 ) } } {#2}
3351   }

```

```

3352     {
3353         \exp_args:Nnf \int_convert_from_base_ten_aux:nnn
3354         { } { \int_eval:n {#1} } {#2}
3355     }
3356 }

```

Here, the idea is to provide a recursive system to deal with the input. The output is build up as argument #1, which is why it starts off empty in the above. At each pass, the value in #2 is checked to see if it is less than the new base (#3). If it is the it is converted directly and the rest of the output is added in. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and the remainder is carried forward to the next round.S

```

3357 \cs_new:Npn \int_convert_from_base_ten_aux:nnn #1#2#3 {
3358     \int_compare:nNnTF {#2} < {#3}
3359     {
3360         \int_convert_number_to_letter:n {#2}
3361         #1
3362     }
3363     {
3364         \exp_args:Nff \int_convert_from_base_ten_aux:nnn
3365         {
3366             \int_convert_number_to_letter:n
3367             { \int_mod:nn {#2} {#3} }
3368             #1
3369         }
3370         { \int_div_truncate:nn {#2} {#3} }
3371         {#3}
3372     }
3373 }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged.

```

3374 \cs_new:Npn \int_convert_number_to_letter:n #1 {
3375     \prg_case_int:nnn { #1 - 9 }
3376     {
3377         { 1 } { A }
3378         { 2 } { B }
3379         { 3 } { C }
3380         { 4 } { D }
3381         { 5 } { E }
3382         { 6 } { F }
3383         { 7 } { G }
3384         { 8 } { H }
3385         { 9 } { I }
3386         { 10 } { J }
3387         { 11 } { K }
3388         { 12 } { L }
3389         { 13 } { M }

```

```

3390      { 14 } { N }
3391      { 15 } { O }
3392      { 16 } { P }
3393      { 17 } { Q }
3394      { 18 } { R }
3395      { 19 } { S }
3396      { 20 } { T }
3397      { 21 } { U }
3398      { 22 } { V }
3399      { 23 } { W }
3400      { 24 } { X }
3401      { 25 } { Y }
3402      { 26 } { Z }
3403    }
3404    {#1}
3405  }

```

(End definition for `\int_convert_from_base_ten:nn`. This function is documented on page 266.)

```

\int_convert_to_base_ten:nn
  \int_convert_to_base_ten_aux:nn
  \int_convert_to_base_ten_aux:nnN
  \int_convert_to_base_ten_aux:N
\int_get_sign_and_digits:n
  \int_get_sign:n
  \int_get_digits:n
\int_get_sign_and_digits_aux:nnN
\int_get_sign_and_digits_aux:oNNN

```

Conversion to base ten means stripping off the sign then iterating through the input one token at a time. The total number is then added up as the code loops.

```

3406 \cs_new:Npn \int_convert_to_base_ten:nn #1#2 {
3407   \int_eval:n
3408   {
3409     \int_get_sign:n {#1}
3410     \exp_args:Nf \int_convert_to_base_ten_aux:nn
3411       { \int_get_digits:n {#1} } {#2}
3412   }
3413 }
3414 \cs_new:Npn \int_convert_to_base_ten_aux:nn #1#2 {
3415   \int_convert_to_base_ten_aux:nnN { 0 } { #2 } #1 \q_nil
3416 }
3417 \cs_new:Npn \int_convert_to_base_ten_aux:nnN #1#2#3 {
3418   \quark_if_nil:NTF #3
3419   {#1}
3420   {
3421     \exp_args:Nf \int_convert_to_base_ten_aux:nnN
3422       { \int_eval:n { #1 * #2 + \int_convert_to_base_ten_aux:N #3 } }
3423       {#2}
3424   }
3425 }

```

The conversion here will take lower or upper case letters and turn them into the appropriate number, hence the two-part nature of the function.

```

3426 \cs_new:Npn \int_convert_to_base_ten_aux:N #1 {
3427   \int_compare:nNnTF { '#1 } < { 58 }
3428   {#1}
3429   {

```

```

3430     \int_eval:n
3431     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
3432   }
3433 }

```

Finding a number and its sign requires dealing with an arbitrary list of + and - symbols. This is done by working through token by token until there is something else at the start of the input. The sign of the input is tracked by the first Boolean used by the auxiliary function.

```

3434 \cs_new:Npn \int_get_sign_and_digits:n #1 {
3435   \int_get_sign_and_digits_aux:nNNN {#1}
3436   \c_true_bool \c_true_bool \c_true_bool
3437 }
3438 \cs_new:Npn \int_get_sign:n #1 {
3439   \int_get_sign_and_digits_aux:nNNN {#1}
3440   \c_true_bool \c_true_bool \c_false_bool
3441 }
3442 \cs_new:Npn \int_get_digits:n #1 {
3443   \int_get_sign_and_digits_aux:nNNN {#1}
3444   \c_true_bool \c_false_bool \c_true_bool
3445 }

```

The auxiliary loops through, finding sign tokens and removing them. The sign itself is carried through as a flag.

```

3446 \cs_new:Npn \int_get_sign_and_digits_aux:nNNN #1#2#3#4 {
3447   \tl_if_head_eq_charcode:fNTF {#1} -
3448   {
3449     \bool_if:NTF #2
3450     {
3451       \int_get_sign_and_digits_aux:oNNN
3452       { \use_none:n #1 } \c_false_bool #3#4
3453     }
3454     {
3455       \int_get_sign_and_digits_aux:oNNN
3456       { \use_none:n #1 } \c_true_bool #3#4
3457     }
3458   }
3459   {
3460     \tl_if_head_eq_charcode:fNTF {#1} +
3461     { \int_get_sign_and_digits_aux:oNNN { \use_none:n #1 } #2#3#4 }
3462     {
3463       \bool_if:NT #3 { \bool_if:NF #2 - }
3464       \bool_if:NT #4 {#1}
3465     }
3466   }
3467 }
3468 \cs_generate_variant:Nn \int_get_sign_and_digits_aux:nNNN { o }

```

(End definition for `\int_convert_to_base_ten:nn`. This function is documented on page [266](#).)

```

\int_from_binary:n
\int_from_hexadecimal:n
\int_from_octal:n
\int_to_binary:n
\int_to_hexadecimal:n
\int_to_octal:n

```

Wrappers around the generic function.

```

3469 \cs_new:Npn \int_from_binary:n #1 {
3470   \int_convert_to_base_ten:nn {#1} { 2 }
3471 }
3472 \cs_new:Npn \int_from_hexadecimal:n #1 {
3473   \int_convert_to_base_ten:nn {#1} { 16 }
3474 }
3475 \cs_new:Npn \int_from_octal:n #1 {
3476   \int_convert_to_base_ten:nn {#1} { 8 }
3477 }
3478 \cs_new:Npn \int_to_binary:n #1 {
3479   \int_convert_from_base_ten:nn {#1} { 2 }
3480 }
3481 \cs_new:Npn \int_to_hexadecimal:n #1 {
3482   \int_convert_from_base_ten:nn {#1} { 16 }
3483 }
3484 \cs_new:Npn \int_to_octal:n #1 {
3485   \int_convert_from_base_ten:nn {#1} { 8 }
3486 }

```

(End definition for `\int_from_binary:n`, `\int_from_hexadecimal:n`, and `\int_from_octal:n`. These functions are documented on page 62.)

```

\int_from_alph:n
\int_from_alph_aux:n
\int_from_alph_aux:nN
\int_from_alph_aux:N

```

The aim here is to iterate through the input, converting one letter at a time to a number. The same approach is also used for base conversion, but this needs a different final auxiliary.

```

3487 \cs_new:Npn \int_from_alph:n #1 {
3488   \int_eval:n
3489   {
3490     \int_get_sign:n {#1}
3491     \exp_args:Nf \int_from_alph_aux:n
3492     { \int_get_digits:n {#1} }
3493   }
3494 }
3495 \cs_new:Npn \int_from_alph_aux:n #1 {
3496   \int_from_alph_aux:nN { 0 } #1 \q_nil
3497 }
3498 \cs_new:Npn \int_from_alph_aux:nN #1#2 {
3499   \quark_if_nil:NTF #2
3500   {#1}
3501   {
3502     \exp_args:Nf \int_from_alph_aux:nN
3503     { \int_eval:n { #1 * 26 + \int_from_alph_aux:N #2 } }
3504   }
3505 }
3506 \cs_new:Npn \int_from_alph_aux:N #1 {
3507   \int_eval:n
3508   { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }
3509 }

```

(End definition for `\int_from_alph:n`. This function is documented on page 62.)

`\int_compare_p:n` Comparison tests using a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. First some notes on the idea behind this. We wish to support writing code like

```
\int_compare_p:n { 5 + \l_tmpa_int != 4 - \l_tmpb_int }
```

In other words, we want to somehow add the missing `\int_eval:w` where required. We can start evaluating from the left using `\int_eval:w`, and we know that since the relation symbols `<`, `>`, `=` and `!` are not allowed in such expressions, they will terminate the expression. Therefore, we first let TeX evaluate this left hand side of the (in)equality.

```
3510 \prg_set_conditional:Npnn \int_compare:n #1{p,TF,T,F}{
3511   \exp_after:wN \int_compare_auxi:w \int_value:w
3512   \int_eval:w #1\q_stop
3513 }
```

Then the next step is to figure out which relation we should use, so we have to somehow get rid of the first evaluation so that we can see what stopped it. `\tex_romannumeral:D` is handy here since its expansion given a non-positive number is `<null>`. We therefore simply check if the first token of the left hand side evaluation is a minus. If not, we insert it and issue `\tex_romannumeral:D`, thereby ridding us of the left hand side evaluation. We do however save it for later.

```
3514 \cs_set:Npn \int_compare_auxi:w #1#2\q_stop{
3515   \exp_after:wN \int_compare_auxii:w \tex_romannumeral:D
3516   \if:w #1- \else: -\fi: #1#2 \q_mark #1#2 \q_stop
3517 }
```

This leaves the first relation symbol in front and assuming the right hand side has been input, at least one other token as well. We support the following forms: `=`, `<`, `>` and the extended `!=`, `==`, `<=` and `>=`. All the extended forms have an extra `=` so we check if that is present as well. Then use specific function to perform the test.

```
3518 \cs_set:Npn \int_compare_auxii:w #1#2#3\q_mark{
3519   \use:c{
3520     int_compare_
3521     #1 \if_meaning:w =#2 = \fi:
3522     :w}
3523 }
```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument. Equality is easy:

```
3524 \cs_set:cpn {int_compare_=:w} #1=#2\q_stop{
3525   \if_int_compare:w #1=\int_eval:w #2 \int_eval_end:
3526   \prg_return_true: \else: \prg_return_false: \fi:
3527 }
```


So is the one using == – we just have to use == in the parameter text.

```

3528 \cs_set:cpn {int_compare_==:w} #1==#2\q_stop{
3529   \if_int_compare:w #1=\int_eval:w #2 \int_eval_end:
3530   \prg_return_true: \else: \prg_return_false: \fi:
3531 }

```

Not equal is just about reversing the truth value.

```

3532 \cs_set:cpn {int_compare_!=:w} #1!=#2\q_stop{
3533   \if_int_compare:w #1=\int_eval:w #2 \int_eval_end:
3534   \prg_return_false: \else: \prg_return_true: \fi:
3535 }

```

Less than and greater than are also straight forward.

```

3536 \cs_set:cpn {int_compare_<:w} #1<#2\q_stop{
3537   \if_int_compare:w #1<\int_eval:w #2 \int_eval_end:
3538   \prg_return_true: \else: \prg_return_false: \fi:
3539 }
3540 \cs_set:cpn {int_compare_>:w} #1>#2\q_stop{
3541   \if_int_compare:w #1>\int_eval:w #2 \int_eval_end:
3542   \prg_return_true: \else: \prg_return_false: \fi:
3543 }

```

The less than or equal operation is just the opposite of the greater than operation. Vice versa for less than or equal.

```

3544 \cs_set:cpn {int_compare_<=:w} #1<=#2\q_stop{
3545   \if_int_compare:w #1>\int_eval:w #2 \int_eval_end:
3546   \prg_return_false: \else: \prg_return_true: \fi:
3547 }
3548 \cs_set:cpn {int_compare_>=:w} #1>=#2\q_stop{
3549   \if_int_compare:w #1<\int_eval:w #2 \int_eval_end:
3550   \prg_return_false: \else: \prg_return_true: \fi:
3551 }

```

(End definition for \int_compare:n. These functions are documented on page 59.)

\int_compare_p:nNn
\int_compare:nNnTF

More efficient but less natural in typing.

```

3552 \prg_set_conditional:Npnn \int_compare:nNn #1#2#3{p}{
3553   \if_int_compare:w \int_eval:w #1 #2 \int_eval:w #3
3554   \int_eval_end:
3555   \prg_return_true: \else: \prg_return_false: \fi:
3556 }
3557 \cs_set_nopar:Npn \int_compare:nNnT #1#2#3 {
3558   \tex_ifnum:D \etex_numexpr:D #1 #2 \etex_numexpr:D #3 \scan_stop:
3559   \tex_expandafter:D \use:n
3560   \tex_else:D
3561   \tex_expandafter:D \use_none:n

```

```

3562 \tex_fi:D
3563 }
3564 \cs_set_nopar:Npn \int_compare:nNnF #1#2#3 {
3565   \tex_ifnum:D \etex_numexpr:D #1 #2 \etex_numexpr:D #3 \scan_stop:
3566     \tex_expandafter:D \use_none:n
3567   \tex_else:D
3568     \tex_expandafter:D \use:n
3569   \tex_fi:D
3570 }
3571 \cs_set_nopar:Npn \int_compare:nNnTF #1#2#3 {
3572   \tex_ifnum:D \etex_numexpr:D #1 #2 \etex_numexpr:D #3 \scan_stop:
3573     \tex_expandafter:D \use_i:nn
3574   \tex_else:D
3575     \tex_expandafter:D \use_ii:nn
3576   \tex_fi:D
3577 }

```

(End definition for `\int_compare:nNn`. These functions are documented on page 59.)

`\int_max:nn` Functions for min, max, and absolute value.

`\int_min:nn`
`\int_abs:n`

```

3578 \cs_set:Npn \int_abs:n #1{
3579   \int_value:w
3580   \if_int_compare:w \int_eval:w #1<\c_zero
3581     -
3582   \fi:
3583   \int_eval:w #1\int_eval_end:
3584 }
3585 \cs_set:Npn \int_max:nn #1#2{
3586   \int_value:w \int_eval:w
3587   \if_int_compare:w
3588     \int_eval:w #1>\int_eval:w #2\int_eval_end:
3589     #1
3590   \else:
3591     #2
3592   \fi:
3593   \int_eval_end:
3594 }
3595 \cs_set:Npn \int_min:nn #1#2{
3596   \int_value:w \int_eval:w
3597   \if_int_compare:w
3598     \int_eval:w #1<\int_eval:w #2\int_eval_end:
3599     #1
3600   \else:
3601     #2
3602   \fi:
3603   \int_eval_end:
3604 }

```

(End definition for `\int_max:nn`. This function is documented on page 56.)

`\int_div_truncate:nn` As `\int_eval:w` rounds the result of a division we also provide a version that truncates
`\int_div_round:nn` the result.
`\int_mod:nn`

Initial version didn't work correctly with eTeX's implementation.

```

3605 %\cs_set:Npn \int_div_truncate_raw:nn #1#2 {
3606 % \int_eval:n{ (2*#1 - #2) / (2* #2) }
3607 %}

```

New version by Heiko:

```

3608 \cs_set:Npn \int_div_truncate:nn #1#2 {
3609   \int_value:w \int_eval:w
3610   \if_int_compare:w \int_eval:w #1 = \c_zero
3611     0
3612   \else:
3613     (#1
3614     \if_int_compare:w \int_eval:w #1 < \c_zero
3615       \if_int_compare:w \int_eval:w #2 < \c_zero
3616         -( #2 +
3617         \else:
3618         +( #2 -
3619         \fi:
3620       \else:
3621       \if_int_compare:w \int_eval:w #2 < \c_zero
3622         +( #2 +
3623         \else:
3624         -( #2 -
3625         \fi:
3626       \fi:
3627       1)/2)
3628     \fi:
3629     /(#2)
3630   \int_eval_end:
3631 }

```

For the sake of completeness:

```

3632 \cs_set:Npn \int_div_round:nn #1#2 {\int_eval:n{(#1)/(#2)}}

```

Finally there's the modulus operation.

```

3633 \cs_set:Npn \int_mod:nn #1#2 {
3634   \int_value:w
3635   \int_eval:w
3636   #1 - \int_div_truncate:nn {#1}{#2} * (#2)
3637   \int_eval_end:
3638 }

```

(End definition for `\int_div_truncate:nn`. This function is documented on page 56.)

`\int_if_odd_p:n` A predicate function.

`\int_if_odd:nTF`
`\int_if_even_p:n`
`\int_if_even:nTF`

```
3639 \prg_set_conditional:Npnn \int_if_odd:n #1 {p,TF,T,F} {  
3640   \if_int_odd:w \int_eval:w #1\int_eval_end:  
3641     \prg_return_true: \else: \prg_return_false: \fi:  
3642 }  
3643 \prg_set_conditional:Npnn \int_if_even:n #1 {p,TF,T,F} {  
3644   \if_int_odd:w \int_eval:w #1\int_eval_end:  
3645     \prg_return_false: \else: \prg_return_true: \fi:  
3646 }
```

(End definition for `\int_if_odd:n`. These functions are documented on page 59.)

`\int_while_do:nn` These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

`\int_until_do:nn`
`\int_do_while:nn`
`\int_do_until:nn`

```
3647 \cs_set:Npn \int_while_do:nn #1#2{  
3648   \int_compare:nT {#1}{#2} \int_while_do:nn {#1}{#2}}  
3649 }  
3650 \cs_set:Npn \int_until_do:nn #1#2{  
3651   \int_compare:nF {#1}{#2} \int_until_do:nn {#1}{#2}}  
3652 }  
3653 \cs_set:Npn \int_do_while:nn #1#2{  
3654   #2 \int_compare:nT {#1}{\int_do_while:nNnn {#1}{#2}}  
3655 }  
3656 \cs_set:Npn \int_do_until:nn #1#2{  
3657   #2 \int_compare:nF {#1}{\int_do_until:nn {#1}{#2}}  
3658 }
```

(End definition for `\int_while_do:nn`. This function is documented on page 60.)

`\int_while_do:nNnn` As above but not using the more natural syntax.

`\int_until_do:nNnn`
`\int_do_while:nNnn`
`\int_do_until:nNnn`

```
3659 \cs_set:Npn \int_while_do:nNnn #1#2#3#4{  
3660   \int_compare:nNnT {#1}{#2}{#3}{#4} \int_while_do:nNnn {#1}{#2}{#3}{#4}}  
3661 }  
3662 \cs_set:Npn \int_until_do:nNnn #1#2#3#4{  
3663   \int_compare:nNnF {#1}{#2}{#3}{#4} \int_until_do:nNnn {#1}{#2}{#3}{#4}}  
3664 }  
3665 \cs_set:Npn \int_do_while:nNnn #1#2#3#4{  
3666   #4 \int_compare:nNnT {#1}{#2}{#3}{\int_do_while:nNnn {#1}{#2}{#3}{#4}}  
3667 }  
3668 \cs_set:Npn \int_do_until:nNnn #1#2#3#4{  
3669   #4 \int_compare:nNnF {#1}{#2}{#3}{\int_do_until:nNnn {#1}{#2}{#3}{#4}}  
3670 }
```

(End definition for `\int_while_do:nNnn`. This function is documented on page 60.)

100.5 Defining constants

`\int_const:Nn` As stated, most constants can be defined as `\tex_chardef:D` or `\tex_mathchardef:D`
`\int_const:cn` but that's engine dependent.

```

3671 \cs_new_protected_nopar:Npn \int_const:Nn #1#2 {
3672   \int_compare:nTF { #2 > \c_minus_one }
3673   {
3674     \int_compare:nTF { #2 > \c_max_register_int }
3675     {
3676       \int_new:N #1
3677       \int_gset:Nn #1 {#2}
3678     }
3679     {
3680       \chk_if_free_cs:N #1
3681       \tex_global:D \tex_mathchardef:D #1 =
3682       \etex_numexpr:D #2 \scan_stop:
3683     }
3684   }
3685   {
3686     \int_new:N #1
3687     \int_gset:Nn #1 {#2}
3688   }
3689 }
3690 \cs_generate_variant:Nn \int_const:Nn { c }

```

(End definition for `\int_const:Nn` and `\int_const:cn`. These functions are documented on page 65.)

`\c_minus_one` And the usual constants, others are still missing. Please, make every constant a real
`\c_zero` constant at least for the moment. We can easily convert things in the end when we have
`\c_one` found what constants are used in critical places and what not.

```

3691 %% \tex_countdef:D \c_minus_one = 10 \scan_stop:
3692 %% \c_minus_one = -1 \scan_stop:          %% in l3basics
3693 %\int_const:Nn \c_zero {0}                %% in l3basics
3694 \int_const:Nn \c_one {1}
3695 \int_const:Nn \c_two {2}
3696 \int_const:Nn \c_three {3}
3697 \int_const:Nn \c_four {4}
3698 \int_const:Nn \c_five {5}
3699 %\int_const:Nn \c_six {6}                %% in l3basics
3700 %\int_const:Nn \c_seven {7}              %% in l3basics
3701 \int_const:Nn \c_eight {8}
3702 \int_const:Nn \c_nine {9}
3703 \int_const:Nn \c_ten {10}
3704 \int_const:Nn \c_eleven {11}
3705 %\int_const:Nn \c_twelve {12}           %% in l3basics
3706 \int_const:Nn \c_thirteen {13}
3707 \int_const:Nn \c_fourteen {14}
3708 \int_const:Nn \c_fifteen {15}

```

`\c_twohundred_fifty_five`
`\c_twohundred_fifty_six`
`\c_thousand`
`\c_ten_thousand`
`\c_ten_thousand_one`
`\c_ten_thousand_two`
`\c_ten_thousand_three`
`\c_ten_thousand_four`
`\c_twenty_thousand`

```

3709 %% \tex_chardef:D \c_sixteen      = 16\scan_stop: %% in l3basics
3710 \int_const:Nn \c_thirty_two {32}

```

The next one may seem a little odd (obviously!) but is useful when dealing with logical operators.

```

3711 \int_const:Nn \c_hundred_one      {101}
3712 \int_const:Nn \c_twohundred_fifty_five{255}
3713 \int_const:Nn \c_twohundred_fifty_six {256}
3714 \int_const:Nn \c_thousand        {1000}
3715 \int_const:Nn \c_ten_thousand    {10000}
3716 \int_const:Nn \c_ten_thousand_one {10001}
3717 \int_const:Nn \c_ten_thousand_two {10002}
3718 \int_const:Nn \c_ten_thousand_three {10003}
3719 \int_const:Nn \c_ten_thousand_four {10004}
3720 \int_const:Nn \c_twenty_thousand {20000}

```

(End definition for `\c_minus_one` and others. These functions are documented on page 66.)

`\c_max_int` The largest number allowed is $2^{31} - 1$

```

3721 \int_const:Nn \c_max_int {2147483647}

```

(End definition for `\c_max_int`. This function is documented on page 66.)

```

\c_int_from_roman_i_int
\c_int_from_roman_v_int
\c_int_from_roman_x_int
\l_int_from_roman_l_int
\c_int_from_roman_c_int
\c_int_from_roman_d_int
\c_int_from_roman_m_int
\c_int_from_roman_I_int
\c_int_from_roman_V_int
\c_int_from_roman_X_int
\c_int_from_roman_L_int
\c_int_from_roman_C_int
\c_int_from_roman_D_int
\c_int_from_roman_M_int

```

Delayed from earlier.

```

3722 \int_const:cn { c_int_from_roman_i_int } { 1 }
3723 \int_const:cn { c_int_from_roman_v_int } { 5 }
3724 \int_const:cn { c_int_from_roman_x_int } { 10 }
3725 \int_const:cn { c_int_from_roman_l_int } { 50 }
3726 \int_const:cn { c_int_from_roman_c_int } { 100 }
3727 \int_const:cn { c_int_from_roman_d_int } { 500 }
3728 \int_const:cn { c_int_from_roman_m_int } { 1000 }
3729 \int_const:cn { c_int_from_roman_I_int } { 1 }
3730 \int_const:cn { c_int_from_roman_V_int } { 5 }
3731 \int_const:cn { c_int_from_roman_X_int } { 10 }
3732 \int_const:cn { c_int_from_roman_L_int } { 50 }
3733 \int_const:cn { c_int_from_roman_C_int } { 100 }
3734 \int_const:cn { c_int_from_roman_D_int } { 500 }
3735 \int_const:cn { c_int_from_roman_M_int } { 1000 }

```

(End definition for `\c_int_from_roman_i_int`.)

Needed from other modules:

```

3736 \int_new:N \g_tl_inline_level_int
3737 \int_new:N \g_prg_inline_level_int

```

100.6 Backwards compatibility

```
3738 \cs_set_eq:NN \intexpr_value:w \int_value:w
3739 \cs_set_eq:NN \intexpr_eval:w \int_eval:w
3740 \cs_set_eq:NN \intexpr_eval_end: \int_eval_end:
3741 \cs_set_eq:NN \if_intexpr_compare:w \if_int_compare:w
3742 \cs_set_eq:NN \if_intexpr_odd:w \if_int_odd:w
3743 \cs_set_eq:NN \if_intexpr_case:w \if_case:w
3744 \cs_set_eq:NN \intexpr_eval:n \int_eval:n
3745
3746 \cs_set_eq:NN \intexpr_compare_p:n \int_compare_p:n
3747 \cs_set_eq:NN \intexpr_compare:nTF \int_compare:nTF
3748 \cs_set_eq:NN \intexpr_compare:nT \int_compare:nT
3749 \cs_set_eq:NN \intexpr_compare:nF \int_compare:nF
3750
3751 \cs_set_eq:NN \intexpr_compare_p:nNn \int_compare_p:nNn
3752 \cs_set_eq:NN \intexpr_compare:nNnTF \int_compare:nNnTF
3753 \cs_set_eq:NN \intexpr_compare:nNnT \int_compare:nNnT
3754 \cs_set_eq:NN \intexpr_compare:nNnF \int_compare:nNnF
3755
3756 \cs_set_eq:NN \intexpr_abs:n \int_abs:n
3757 \cs_set_eq:NN \intexpr_max:nn \int_max:nn
3758 \cs_set_eq:NN \intexpr_min:nn \int_min:nn
3759
3760 \cs_set_eq:NN \intexpr_div_truncate:nn \int_div_truncate:nn
3761 \cs_set_eq:NN \intexpr_div_round:nn \int_div_round:nn
3762 \cs_set_eq:NN \intexpr_mod:nn \int_mod:nn
3763
3764 \cs_set_eq:NN \intexpr_if_odd_p:n \int_if_odd_p:n
3765 \cs_set_eq:NN \intexpr_if_odd:nTF \int_if_odd:nTF
3766 \cs_set_eq:NN \intexpr_if_odd:nT \int_if_odd:nT
3767 \cs_set_eq:NN \intexpr_if_odd:nF \int_if_odd:nF
3768
3769 \cs_set_eq:NN \intexpr_if_even_p:n \int_if_even_p:n
3770 \cs_set_eq:NN \intexpr_if_even:nTF \int_if_even:nTF
3771 \cs_set_eq:NN \intexpr_if_even:nT \int_if_even:nT
3772 \cs_set_eq:NN \intexpr_if_even:nF \int_if_even:nF
3773
3774 \cs_set_eq:NN \intexpr_while_do:nn \int_while_do:nn
3775 \cs_set_eq:NN \intexpr_until_do:nn \int_until_do:nn
3776 \cs_set_eq:NN \intexpr_do_while:nn \int_do_while:nn
3777 \cs_set_eq:NN \intexpr_do_until:nn \int_do_until:nn
3778
3779 \cs_set_eq:NN \intexpr_while_do:nNnn \int_while_do:nNnn
3780 \cs_set_eq:NN \intexpr_until_do:nNnn \int_until_do:nNnn
3781 \cs_set_eq:NN \intexpr_do_while:nNnn \int_do_while:nNnn
3782 \cs_set_eq:NN \intexpr_do_until:nNnn \int_do_until:nNnn
3783 </initex | package>
```

101 l3skip implementation

We start by ensuring that the required packages are loaded.

```
3784 <*package>
3785 \ProvidesExplPackage
3786   {\filename}{\filedate}{\fileversion}{\filedescription}
3787 \package_check_loaded_expl:
3788 </package>
3789 <*initex | package>
```

101.1 Skip registers

\skip_new:N Allocation of a new internal registers.

\skip_new:c

```
3790 <*initex>
3791 \alloc_new:nnnN {skip} \c_zero \c_max_register_int \tex_skipdef:D
3792 </initex>
3793 <*package>
3794 \cs_new_protected_nopar:Npn \skip_new:N #1 {
3795   \chk_if_free_cs:N #1
3796   \newskip #1
3797 }
3798 </package>
3799 \cs_generate_variant:Nn \skip_new:N {c}
```

(End definition for `\skip_new:N` and `\skip_new:c`. These functions are documented on page 68.)

\skip_set:Nn Setting skips is again something that I would like to make uniform at the moment to get
\skip_set:cn a better overview.

\skip_gset:Nn

\skip_gset:cn

```
3800 \cs_new_protected_nopar:Npn \skip_set:Nn #1#2 {
3801   #1 \etex_glueexpr:D #2 \scan_stop:
3802 <*check>
3803 \chk_local_or_pref_global:N #1
3804 </check>
3805 }
3806 \cs_new_protected_nopar:Npn \skip_gset:Nn {
3807 <*check>
3808   \pref_global_chk:
3809 </check>
3810 <-check> \pref_global:D
3811   \skip_set:Nn
3812 }
3813 \cs_generate_variant:Nn \skip_set:Nn {cn}
3814 \cs_generate_variant:Nn \skip_gset:Nn {cn}
```

(End definition for `\skip_set:Nn`. This function is documented on page 69.)


```

\skip_zero:N Reset the register to zero.
\skip_gzero:N
\skip_zero:c
\skip_gzero:c
3815 \cs_new_protected_nopar:Npn \skip_zero:N #1{
3816   #1\c_zero_skip \scan_stop:
3817   <*check>
3818   \chk_local_or_pref_global:N #1
3819   </check>
3820 }
3821 \cs_new_protected_nopar:Npn \skip_gzero:N {

```

We make sure that a local variable is not updated globally by changing the internal test (i.e. `\chk_local_or_pref_global:N`) before making the assignment. This is done by `\pref_global_chk:` which also issues the necessary `\pref_global:D`. This is not very efficient, but this code will be only included for debugging purposes. Using `\pref_global:D` in front of the local function is better in the production versions.

```

3822 <*check>
3823 \pref_global_chk:
3824 </check>
3825 <-check> \pref_global:D
3826 \skip_zero:N
3827 }
3828 \cs_generate_variant:Nn \skip_zero:N {c}
3829 \cs_generate_variant:Nn \skip_gzero:N {c}

```

(End definition for `\skip_zero:N`. This function is documented on page 69.)

```

\skip_add:Nn Adding and subtracting to and from <skip>s
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn
\skip_sub:Nn
\skip_gsub:Nn
3830 \cs_new_protected_nopar:Npn \skip_add:Nn #1#2 {

```

We need to say by in case the first argument is a register accessed by its number, e.g., `\skip23`.

```

3831 \tex_advance:D#1 by \etex_glueexpr:D #2 \scan_stop:
3832 <*check>
3833 \chk_local_or_pref_global:N #1
3834 </check>
3835 }
3836 \cs_generate_variant:Nn \skip_add:Nn {cn}

3837 \cs_new_protected_nopar:Npn \skip_sub:Nn #1#2{
3838   \tex_advance:D #1 -\etex_glueexpr:D #2 \scan_stop:
3839   <*check>
3840   \chk_local_or_pref_global:N #1
3841   </check>
3842 }

3843 \cs_new_protected_nopar:Npn \skip_gadd:Nn {
3844   <*check>

```

```

3845 \pref_global_chk:
3846 </check>
3847 <-check> \pref_global:D
3848 \skip_add:Nn
3849 }
3850 \cs_generate_variant:Nn \skip_gadd:Nn {cn}

3851 \cs_new_nopar:Npn \skip_gsub:Nn {
3852 <*check>
3853 \pref_global_chk:
3854 </check>
3855 <-check> \pref_global:D
3856 \skip_sub:Nn
3857 }

```

(End definition for `\skip_add:Nn`. This function is documented on page 69.)

`\skip_horizontal:N` Inserting skips.

```

\skip_horizontal:c
\skip_horizontal:n
\skip_vertical:N
\skip_vertical:c
\skip_vertical:n
3858 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
3859 \cs_generate_variant:Nn \skip_horizontal:N {c}

3860 \cs_new_nopar:Npn \skip_horizontal:n #1 {
3861 \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop:
3862 }
3863 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
3864 \cs_generate_variant:Nn \skip_vertical:N {c}
3865 \cs_new_nopar:Npn \skip_vertical:n #1 {
3866 \skip_vertical:N \etex_glueexpr:D #1 \scan_stop:
3867 }

```

(End definition for `\skip_horizontal:N`. This function is documented on page 70.)

`\skip_use:N` Here is how skip registers are accessed:

```

\skip_use:c
3868 \cs_new_eq:NN \skip_use:N \tex_the:D
3869 \cs_generate_variant:Nn \skip_use:N {c}

```

(End definition for `\skip_use:N`. This function is documented on page 69.)

`\skip_show:N` Diagnostics.

```

\skip_show:c
3870 \cs_new_eq:NN \skip_show:N \kernel_register_show:N
3871 \cs_generate_variant:Nn \skip_show:N {c}

```

(End definition for `\skip_show:N`. This function is documented on page 69.)

`\skip_eval:n` Evaluating a calc expression. This is expandable and works inside an “x” type of argument.

```

3872 \cs_new_nopar:Npn \skip_eval:n #1 {
3873 \tex_the:D \etex_glueexpr:D #1 \scan_stop:
3874 }

```

(End definition for `\skip_eval:n`. This function is documented on page 70.)

`\l_tmpa_skip` We provide three local and two global scratch registers, maybe we need more or less.

```

\l_tmpb_skip
\l_tmpc_skip
\g_tmpa_skip
\g_tmpb_skip
3875 %%\chk_if_free_cs:N \l_tmpa_skip
3876 %%\tex_skipdef:D\l_tmpa_skip 255 %currently taken up by \skip@
3877 \skip_new:N \l_tmpa_skip
3878 \skip_new:N \l_tmpb_skip
3879 \skip_new:N \l_tmpc_skip
3880 \skip_new:N \g_tmpa_skip
3881 \skip_new:N \g_tmpb_skip

```

(End definition for `\l_tmpa_skip`. This function is documented on page 71.)

`\c_zero_skip`
`\c_max_skip`

```

3882 <*\package>
3883 \skip_new:N \c_zero_skip
3884 \skip_set:Nn \c_zero_skip {0pt}
3885 \skip_new:N \c_max_skip
3886 \skip_set:Nn \c_max_skip {16383.99999pt}
3887 </!\package>
3888 <*\initex>
3889 \cs_set_eq:NN \c_zero_skip \z@
3890 \cs_set_eq:NN \c_max_skip \maxdimen
3891 </!\initex>

```

(End definition for `\c_zero_skip`. This function is documented on page 71.)

`\skip_if_infinite_glue_p:n` With ϵ -TeX we all of a sudden get access to a lot information we should otherwise consider
`\skip_if_infinite_glue:nTF` ourselves lucky to get. One is the stretch and shrink components of a skip register
and the order of those components. `\skip_if_infinite_glue:nTF` tests it directly by
looking at the stretch and shrink order. If either of the predicate functions return `<true>`
`\bool_if:nTF` will return `<true>` and the logic test will take the true branch.

```

3892 \prg_new_conditional:Nnn \skip_if_infinite_glue:n {p,TF,T,F} {
3893   \bool_if:nTF {
3894     \int_compare_p:nNn {\etex_gluestretchorder:D #1 } > \c_zero ||
3895     \int_compare_p:nNn {\etex_glueshrinkorder:D #1 } > \c_zero
3896   } {\prg_return_true:} {\prg_return_false:}
3897 }

```

(End definition for `\skip_if_infinite_glue_p:n`. This function is documented on page 70.)

`\skip_split_finite_else_action:nnNN` This macro is useful when performing error checking in certain circumstances. If the
`<skip>` register holds finite glue it sets #3 and #4 to the stretch and shrink component
resp. If it holds infinite glue set #3 and #4 to zero and issue the special action #2 which
is probably an error message. Assignments are global.

```

3898 \cs_new_nopar:Npn \skip_split_finite_else_action:nnNN #1#2#3#4{
3899   \skip_if_infinite_glue:nTF {#1}
3900   {
3901     #3 = \c_zero_skip
3902     #4 = \c_zero_skip
3903     #2
3904   }
3905   {
3906     #3 = \etex_gluestretch:D #1 \scan_stop:
3907     #4 = \etex_glueshrink:D #1 \scan_stop:
3908   }
3909 }

```

(End definition for `\skip_split_finite_else_action:nnNN`. This function is documented on page 70.)

101.2 Dimen registers

`\dim_new:N` Allocating $\langle dim \rangle$ registers...
`\dim_new:c`

```

3910 \<initex>
3911 \alloc_new:nnnN {dim} \c_zero \c_max_register_int \tex_dimendef:D
3912 \</initex>
3913 \<*package>
3914 \cs_new_protected_nopar:Npn \dim_new:N #1 {
3915   \chk_if_free_cs:N #1
3916   \newdimen #1
3917 }
3918 \</package>
3919 \cs_generate_variant:Nn \dim_new:N {c}

```

(End definition for `\dim_new:N` and `\dim_new:c`. These functions are documented on page 71.)

`\dim_set:Nn` We add `\dim_eval:n` in order to allow simple arithmetic and a space just for those using
`\dim_set:cn` `\dimen1` or alike. See OR!
`\dim_set:Nc`

```

3920 \cs_new_protected_nopar:Npn \dim_set:Nn #1#2 {
3921   #1~ \etex_dimexpr:D #2 \scan_stop:
3922 }
3923 \cs_generate_variant:Nn \dim_set:Nn {cn,Nc}

3924 \cs_new_protected_nopar:Npn \dim_gset:Nn { \pref_global:D \dim_set:Nn }
3925 \cs_generate_variant:Nn \dim_gset:Nn {cn,Nc,cc}

```

(End definition for `\dim_set:Nn`. This function is documented on page 72.)

`\dim_set_max:Nn` Setting maximum and minimum values is simply a case of so build-in comparison. This
`\dim_set_max:cn` only applies to dimensions as skips are not ordered.
`\dim_set_min:Nn`

```

3926 \cs_new_protected_nopar:Npn \dim_set_max:Nn #1#2 {
\dim_set_max:cn
\dim_gset_max:Nn
\dim_gset_max:cn
\dim_gset_min:Nn
\dim_gset_min:cn

```

```

3927 \dim_compare:nNnT {#1} < {#2} { \dim_set:Nn #1 {#2} }
3928 }
3929 \cs_generate_variant:Nn \dim_set_max:Nn { c }
3930 \cs_new_protected_nopar:Npn \dim_set_min:Nn #1#2 {
3931 \dim_compare:nNnT {#1} > {#2} { \dim_set:Nn #1 {#2} }
3932 }
3933 \cs_generate_variant:Nn \dim_set_min:Nn { c }
3934 \cs_new_protected_nopar:Npn \dim_gset_max:Nn #1#2 {
3935 \dim_compare:nNnT {#1} < {#2} { \dim_gset:Nn #1 {#2} }
3936 }
3937 \cs_generate_variant:Nn \dim_gset_max:Nn { c }
3938 \cs_new_protected_nopar:Npn \dim_gset_min:Nn #1#2 {
3939 \dim_compare:nNnT {#1} > {#2} { \dim_gset:Nn #1 {#2} }
3940 }
3941 \cs_generate_variant:Nn \dim_gset_min:Nn { c }

```

(End definition for `\dim_set_max:Nn`. This function is documented on page 72.)

`\dim_zero:N` Resetting.

```

\dim_gzero:N
\dim_zero:c
\dim_gzero:c
3942 \cs_new_protected_nopar:Npn \dim_zero:N #1 { #1\c_zero_skip }
3943 \cs_generate_variant:Nn \dim_zero:N {c}

3944 \cs_new_protected_nopar:Npn \dim_gzero:N { \pref_global:D \dim_zero:N }
3945 \cs_generate_variant:Nn \dim_gzero:N {c}

```

(End definition for `\dim_zero:N`. This function is documented on page 71.)

`\dim_add:Nn` Addition.

```

\dim_add:cn
\dim_add:Nc
3946 \cs_new_protected_nopar:Npn \dim_add:Nn #1#2{

```

`\dim_gadd:Nn` We need to say by in case the first argument is a register accessed by its number, e.g.,
`\dim_gadd:cn` `\dimen23`.

```

3947 \tex_advance:D#1 by \etex_dimexpr:D #2 \scan_stop:
3948 }
3949 \cs_generate_variant:Nn \dim_add:Nn {cn,Nc}

3950 \cs_new_protected_nopar:Npn \dim_gadd:Nn { \pref_global:D \dim_add:Nn }
3951 \cs_generate_variant:Nn \dim_gadd:Nn {cn}

```

(End definition for `\dim_add:Nn`. This function is documented on page 72.)

`\dim_sub:Nn` Subtracting.

```

\dim_sub:cn
\dim_sub:Nc
3952 \cs_new_protected_nopar:Npn \dim_sub:Nn #1#2 { \tex_advance:D#1-#2\scan_stop: }
3953 \cs_generate_variant:Nn \dim_sub:Nn {cn,Nc}

\dim_gsub:Nn
\dim_gsub:cn
3954 \cs_new_protected_nopar:Npn \dim_gsub:Nn { \pref_global:D \dim_sub:Nn }
3955 \cs_generate_variant:Nn \dim_gsub:Nn {cn}

```

(End definition for `\dim_sub:Nn`. This function is documented on page 73.)

`\dim_use:N` Accessing a $\langle dim \rangle$.

`\dim_use:c`

```

3956 \cs_new_eq:NN \dim_use:N \tex_the:D
3957 \cs_generate_variant:Nn \dim_use:N {c}

```

(End definition for `\dim_use:N`. This function is documented on page 73.)

`\dim_show:N` Diagnostics.

`\dim_show:c`

```

3958 \cs_new_eq:NN \dim_show:N \kernel_register_show:N
3959 \cs_generate_variant:Nn \dim_show:N {c}

```

(End definition for `\dim_show:N`. This function is documented on page 73.)

`\l_tmpa_dim` Some scratch registers.

`\l_tmpb_dim`

```

3960 \dim_new:N \l_tmpa_dim
3961 \dim_new:N \l_tmpb_dim
3962 \dim_new:N \l_tmpc_dim
3963 \dim_new:N \l_tmpd_dim
3964 \dim_new:N \g_tmpa_dim
3965 \dim_new:N \g_tmpb_dim

```

(End definition for `\l_tmpa_dim`. This function is documented on page 75.)

`\c_zero_dim` Just aliases.

`\c_max_dim`

```

3966 \cs_new_eq:NN \c_zero_dim \c_zero_skip
3967 \cs_new_eq:NN \c_max_dim \c_max_skip

```

(End definition for `\c_zero_dim`. This function is documented on page 75.)

`\dim_eval:n` Evaluating a calc expression. This is expandable and works inside an “x” type of argument.

```

3968 \cs_new_nopar:Npn \dim_eval:n #1 {
3969   \tex_the:D \etex_dimexpr:D #1 \scan_stop:
3970 }

```

(End definition for `\dim_eval:n`. This function is documented on page 73.)

`\if_dim:w` Primitives.

`\dim_value:w`

```

3971 \cs_new_eq:NN \if_dim:w \tex_ifdim:D
3972 \cs_set_eq:NN \dim_value:w \tex_number:D
3973 \cs_set_eq:NN \dim_eval:w \etex_dimexpr:D
3974 \cs_set_protected:Npn \dim_eval_end: {\tex_relax:D}

```

(End definition for `\if_dim:w` and others. These functions are documented on page ??.)

`\dim_compare_p:nNn`
`\dim_compare:nNnTF`

```

3975 \prg_new_conditional:Nnn \dim_compare:nNn {p,TF,T,F} {
3976   \if_dim:w \etex_dimexpr:D #1 #2 \etex_dimexpr:D #3 \scan_stop:
3977     \prg_return_true: \else: \prg_return_false: \fi:
3978 }

```

(End definition for `\dim_compare_p:nNn`. This function is documented on page 74.)

`\dim_compare_p:n`
`\dim_compare:nTF`

[This code plus comments lifted directly from the `\int_compare:nTF` function.] Comparison tests using a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. First some notes on the idea behind this. We wish to support writing code like

```
\dim_compare_p:n { 5 + \l_tmpa_dim != 4 - \l_tmpb_dim }
```

In other words, we want to somehow add the missing `\dim_eval:w` where required. We can start evaluating from the left using `\dim:w`, and we know that since the relation symbols `<`, `>`, `=` and `!` are not allowed in such expressions, they will terminate the expression. Therefore, we first let `TeX` evaluate this left hand side of the (in)equality.

```

3979 \prg_new_conditional:Npnn \dim_compare:n #1 {p,TF,T,F} {
3980   \exp_after:wN \dim_compare_auxi:w \dim_value:w
3981     \dim_eval:w #1 \q_stop
3982 }

```

Then the next step is to figure out which relation we should use, so we have to somehow get rid of the first evaluation so that we can see what stopped it. `\tex_romannumeral:D` is handy here since its expansion given a non-positive number is `<null>`. We therefore simply check if the first token of the left hand side evaluation is a minus. If not, we insert it and issue `\tex_romannumeral:D`, thereby ridding us of the left hand side evaluation. We do however save it for later.

```

3983 \cs_new:Npn \dim_compare_auxi:w #1#2 \q_stop {
3984   \exp_after:wN \dim_compare_auxii:w \tex_romannumeral:D
3985   \if:w #1- \else: -\fi: #1#2 \q_mark #1#2 \q_stop
3986 }

```

This leaves the first relation symbol in front and assuming the right hand side has been input, at least one other token as well. We support the following forms: `=`, `<`, `>` and the extended `!=`, `==`, `<=` and `>=`. All the extended forms have an extra `=` so we check if that is present as well. Then use specific function to perform the test.

```

3987 \cs_new:Npn \dim_compare_auxii:w #1#2#3\q_mark{
3988   \use:c{
3989     dim_compare_ #1 \if_meaning:w =#2 = \fi:
3990     :w}
3991 }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument. Equality is easy:

```

3992 \cs_new:cpn {dim_compare_=:w} #1 = #2 \q_stop {
3993   \if_dim:w #1 sp = \dim_eval:w #2 \dim_eval_end:
3994   \prg_return_true: \else: \prg_return_false: \fi:
3995 }

```

So is the one using == – we just have to use == in the parameter text.

```

3996 \cs_new:cpn {dim_compare_==:w} #1 == #2 \q_stop {
3997   \if_dim:w #1 sp = \dim_eval:w #2 \dim_eval_end:
3998   \prg_return_true: \else: \prg_return_false: \fi:
3999 }

```

Not equal is just about reversing the truth value.

```

4000 \cs_new:cpn {dim_compare_!=:w} #1 != #2 \q_stop {
4001   \if_dim:w #1 sp = \dim_eval:w #2 \dim_eval_end:
4002   \prg_return_false: \else: \prg_return_true: \fi:
4003 }

```

Less than and greater than are also straight forward.

```

4004 \cs_new:cpn {dim_compare<:w} #1 < #2 \q_stop {
4005   \if_dim:w #1 sp < \dim_eval:w #2 \dim_eval_end:
4006   \prg_return_true: \else: \prg_return_false: \fi:
4007 }
4008 \cs_new:cpn {dim_compare>:w} #1 > #2 \q_stop {
4009   \if_dim:w #1 sp > \dim_eval:w #2 \dim_eval_end:
4010   \prg_return_true: \else: \prg_return_false: \fi:
4011 }

```

The less than or equal operation is just the opposite of the greater than operation. Vice versa for less than or equal.

```

4012 \cs_new:cpn {dim_compare<=:w} #1 <= #2 \q_stop {
4013   \if_dim:w #1 sp > \dim_eval:w #2 \dim_eval_end:
4014   \prg_return_false: \else: \prg_return_true: \fi:
4015 }
4016 \cs_new:cpn {dim_compare>=:w} #1 >= #2 \q_stop {
4017   \if_dim:w #1 sp < \dim_eval:w #2 \dim_eval_end:
4018   \prg_return_false: \else: \prg_return_true: \fi:
4019 }

```

(End definition for `\dim_compare_p:n`. This function is documented on page 74.)

`\dim_while_do:nNnn` while_do and do_while functions for dimensions. Same as for the `int` type only the names have changed.

`\dim_until_do:nNnn`

`\dim_do_while:nNnn`

`\dim_do_until:nNnn`

```

4020 \cs_new_nopar:Npn \dim_while_do:nNnn #1#2#3#4{

```



```

4021 \dim_compare:nNnT {#1}#2{#3}{#4 \dim_while_do:nNnn {#1}#2{#3}{#4}}
4022 }
4023 \cs_new_nopar:Npn \dim_until_do:nNnn #1#2#3#4{
4024 \dim_compare:nNnF {#1}#2{#3}{#4 \dim_until_do:nNnn {#1}#2{#3}{#4}}
4025 }
4026 \cs_new_nopar:Npn \dim_do_while:nNnn #1#2#3#4{
4027 #4 \dim_compare:nNnT {#1}#2{#3}{\dim_do_while:nNnn {#1}#2{#3}{#4}}
4028 }
4029 \cs_new_nopar:Npn \dim_do_until:nNnn #1#2#3#4{
4030 #4 \dim_compare:nNnF {#1}#2{#3}{\dim_do_until:nNnn {#1}#2{#3}{#4}}
4031 }

```

(End definition for `\dim_while_do:nNnn`. This function is documented on page 74.)

101.3 Muskips

`\muskip_new:N` And then we add muskips.

```

4032 <*initex>
4033 \alloc_new:nnnN {muskip} \c_zero \c_max_register_int \tex_muskipdef:D
4034 </initex>
4035 <*package>
4036 \cs_new_protected_nopar:Npn \muskip_new:N #1 {
4037 \chk_if_free_cs:N #1
4038 \newmuskip #1
4039 }
4040 </package>

```

(End definition for `\muskip_new:N`. This function is documented on page 75.)

`\muskip_set:Nn` `\muskip_gset:Nn` `\muskip_add:Nn` `\muskip_gadd:Nn` `\muskip_sub:Nn` `\muskip_gsub:Nn` Simple functions for muskips.

```

4041 \cs_new_protected_nopar:Npn \muskip_set:Nn#1#2{#1\etex_muexpr:D#2\scan_stop:}
4042 \cs_new_protected_nopar:Npn \muskip_gset:Nn{\pref_global:D\muskip_set:Nn}
4043 \cs_new_protected_nopar:Npn \muskip_add:Nn#1#2{\tex_advance:D#1\etex_muexpr:D#2\scan_stop:}
4044 \cs_new_protected_nopar:Npn \muskip_gadd:Nn{\pref_global:D\muskip_add:Nn}
4045 \cs_new_protected_nopar:Npn \muskip_sub:Nn#1#2{\tex_advance:D#1-\etex_muexpr:D#2\scan_stop:}
4046 \cs_new_protected_nopar:Npn \muskip_gsub:Nn{\pref_global:D\muskip_sub:Nn}

```

(End definition for `\muskip_set:Nn`. This function is documented on page 75.)

`\muskip_use:N` Accessing a `<muskip>`.

```

4047 \cs_new_eq:NN \muskip_use:N \tex_the:D

```

(End definition for `\muskip_use:N`. This function is documented on page 75.)

`\muskip_show:N`

```

4048 \cs_new_eq:NN \muskip_show:N \kernel_register_show:N

```

(End definition for `\muskip_show:N`. This function is documented on page 76.)

```

4049 </initex | package>

```

102 l3tl implementation

We start by ensuring that the required packages are loaded.

```

4050 <*package>
4051 \ProvidesExplPackage
4052   {\filename}{\filedate}{\fileversion}{\filedescription}
4053 \package_check_loaded_expl:
4054 </package>

4055 <*initex | package>

```

A token list variable is a control sequence that holds tokens. The interface is similar to that for token registers, but beware that the behavior vis á vis `\cs_set_nopar:Npx` etc. ... is different. (You see this comes from Denys' implementation.)

102.1 Functions

`\tl_new:N` We provide one allocation function (which checks that the name is not used) and two clear functions that locally or globally clear the token list. The allocation function has two arguments to specify an initial value. This is the only way to give values to constants.

```

\tl_new:c
\tl_new:Nn
\tl_new:cn
\tl_new:Nx
4056 \cs_new_protected:Npn \tl_new:Nn #1#2{
4057   \chk_if_free_cs:N #1

```

If checking we don't allow constants to be defined.

```

4058 <*check>
4059   \chk_var_or_const:N #1
4060 </check>

```

Otherwise any variable type is allowed.

```

4061   \tl_gset:Nn #1 {#2}
4062 }
4063 \cs_generate_variant:Nn \tl_new:Nn {cn}
4064 \cs_new_protected:Npn \tl_new:Nx #1#2{
4065   \chk_if_free_cs:N #1
4066 <check> \chk_var_or_const:N #1
4067   \tl_gset:Nx #1 {#2}
4068 }
4069 \cs_new_protected_nopar:Npn \tl_new:N #1{\tl_new:Nn #1{}}
4070 \cs_new_protected_nopar:Npn \tl_new:c #1{\tl_new:cn {#1}{}}

```

(End definition for `\tl_new:N`. This function is documented on page 76.)

`\tl_const:Nn` For creating constant token lists: there is not actually anything here that cannot be achieved using `\tl_new:N` and `\tl_set:Nn`

```

4071 \cs_new_protected:Npn \tl_const:Nn #1#2 {
4072   \tl_new:N #1
4073   \tl_gset:Nn #1 {#2}
4074 }

```

(End definition for `\tl_const:Nn`. This function is documented on page 76.)

`\tl_use:N` Perhaps this should just be enabled when checking?
`\tl_use:c`

```

4075 \cs_new_nopar:Npn \tl_use:N #1 {
4076   \if_meaning:w #1 \tex_relax:D

```

If `\tl var.` equals `\tex_relax:D` it is probably stemming from a `\cs:w... \cs_end:` that was created by mistake somewhere.

```

4077   \msg_kernel_bug:x {Token~list~variable~ '\token_to_str:N #1'~
4078                     has~ an~ erroneous~ structure!}
4079   \else:
4080     \exp_after:wN #1
4081     \fi:
4082   }
4083   \cs_generate_variant:Nn \tl_use:N {c}

```

(End definition for `\tl_use:N`. This function is documented on page 76.)

`\tl_show:N` Showing a `\tl var.` is just `\show`ing it and I don't really care about checking that it's
`\tl_show:c` malformed at this stage.
`\tl_show:n`

```

4084 \cs_new_nopar:Npn \tl_show:N #1 { \cs_show:N #1 }
4085 \cs_generate_variant:Nn \tl_show:N {c}
4086 \cs_set_eq:NN \tl_show:n \etex_showtokens:D

```

(End definition for `\tl_show:N`, `\tl_show:c`, and `\tl_show:n`. These functions are documented on page 77.)

`\tl_set:Nn` By using `\exp_not:n` token list variables can contain # tokens.

```

\tl_set:NV
\tl_set:Nv
\tl_set:Nx
\tl_set:Nf
\tl_set:Nc
\tl_set:cV
\tl_set:cv
\tl_set:co
\tl_set:cx
\tl_gset:Nn
\tl_gset:NV
\tl_gset:Nv
\tl_gset:No
\tl_gset:Nf
\tl_gset:Nx
\tl_gset:cV
\tl_gset:cv
\tl_gset:co
\tl_gset:cx

```

```

4101 \cs_generate_variant:Nn \tl_set:Nn { No }
4102 \cs_generate_variant:Nn \tl_set:Nn { Nf }
4103 \cs_generate_variant:Nn \tl_set:Nn { cV }
4104 \cs_generate_variant:Nn \tl_set:Nn { c }
4105 \cs_generate_variant:Nn \tl_set:Nn { cv }
4106 \cs_generate_variant:Nn \tl_set:Nn { co }
4107 \cs_generate_variant:Nn \tl_set:Nx { c }
4108 \cs_generate_variant:Nn \tl_gset:Nn { NV }
4109 \cs_generate_variant:Nn \tl_gset:Nn { Nv }
4110 \cs_generate_variant:Nn \tl_gset:Nn { No }
4111 \cs_generate_variant:Nn \tl_gset:Nn { Nf }
4112 \cs_generate_variant:Nn \tl_gset:Nn { c }
4113 \cs_generate_variant:Nn \tl_gset:Nn { cV }
4114 \cs_generate_variant:Nn \tl_gset:Nn { cv }
4115 \cs_generate_variant:Nn \tl_gset:Nx { c }

```

(End definition for `\tl_set:Nn`. This function is documented on page 77.)

`\tl_set_eq:NN` For setting token list variables equal to each other. First checking:
`\tl_set_eq:Nc`
`\tl_set_eq:cN`
`\tl_set_eq:cc`
`\tl_gset_eq:NN`
`\tl_gset_eq:Nc`
`\tl_gset_eq:cN`
`\tl_gset_eq:cc`

```

4116 <*check>
4117 \cs_new_protected_nopar:Npn \tl_set_eq:NN #1#2{
4118   \chk_exist_cs:N #1 \cs_set_eq:NN #1#2
4119   \chk_local_or_pref_global:N #1 \chk_var_or_const:N #2
4120 }
4121 \cs_new_protected_nopar:Npn \tl_gset_eq:NN #1#2{
4122   \chk_exist_cs:N #1 \cs_gset_eq:NN #1#2
4123   \chk_global:N #1 \chk_var_or_const:N #2
4124 }
4125 </check>

```

Non-checking versions are easy.

```

4126 <!*!check>
4127 \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
4128 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
4129 </!check>

```

The rest again with the expansion module.

```

4130 \cs_generate_variant:Nn \tl_set_eq:NN {Nc,c,cc}
4131 \cs_generate_variant:Nn \tl_gset_eq:NN {Nc,c,cc}

```

(End definition for `\tl_set_eq:NN`. This function is documented on page 79.)

`\tl_clear:N` Clearing a token list variable.
`\tl_clear:c`
`\tl_gclear:N`
`\tl_gclear:c`

```

4132 \cs_new_protected_nopar:Npn \tl_clear:N #1{\tl_set_eq:NN #1\c_empty_tl}
4133 \cs_generate_variant:Nn \tl_clear:N {c}
4134 \cs_new_protected_nopar:Npn \tl_gclear:N #1{\tl_gset_eq:NN #1\c_empty_tl}
4135 \cs_generate_variant:Nn \tl_gclear:N {c}

```

(End definition for `\tl_clear:N`. This function is documented on page 77.)

`\tl_clear_new:N` These macros check whether a token list exists. If it does it is cleared, if it doesn't it is allocated.
`\tl_clear_new:c`

```

4136 <*check>
4137 \cs_new_protected_nopar:Npn \tl_clear_new:N #1{
4138   \chk_var_or_const:N #1
4139   \if_predicate:w \cs_if_exist_p:N #1
4140     \tl_clear:N #1
4141   \else:
4142     \tl_new:N #1
4143   \fi:
4144 }
4145 </check>
4146 <-check>\cs_new_eq:NN \tl_clear_new:N \tl_clear:N
4147 \cs_generate_variant:Nn \tl_clear_new:N {c}

```

(End definition for `\tl_clear_new:N`. This function is documented on page 77.)

`\tl_gclear_new:N` These are the global versions of the above.
`\tl_gclear_new:c`

```

4148 <*check>
4149 \cs_new_protected_nopar:Npn \tl_gclear_new:N #1{
4150   \chk_var_or_const:N #1
4151   \if_predicate:w \cs_if_exist_p:N #1
4152     \tl_gclear:N #1
4153   \else:
4154     \tl_new:N #1
4155   \fi:}
4156 </check>
4157 <-check>\cs_new_eq:NN \tl_gclear_new:N \tl_gclear:N
4158 \cs_generate_variant:Nn \tl_gclear_new:N {c}

```

(End definition for `\tl_gclear_new:N`. This function is documented on page 77.)

`\tl_put_right:Nn` Adding to one end of a token list is done partially using hand tuned functions for performance reasons.
`\tl_put_right:Nv`
`\tl_put_right:Nx`
`\tl_put_right:cn`
`\tl_put_right:cV`
`\tl_put_right:cv`
`\tl_put_right:cx`
`\tl_gput_right:Nn`
`\tl_gput_right:Nv`
`\tl_gput_right:Nx`
`\tl_gput_right:cn`
`\tl_gput_right:cV`
`\tl_gput_right:cv`
`\tl_gput_right:co`
`\tl_gput_right:cx`

```

4159 \cs_new_protected:Npn \tl_put_right:Nn #1#2 {
4160   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} }
4161 }
4162 \cs_new_protected:Npn \tl_put_right:Nv #1#2 {
4163   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 }
4164 }
4165 \cs_new_protected:Npn \tl_put_right:Nx #1#2 {
4166   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:v {#2} }
4167 }
4168 \cs_new_protected:Npn \tl_put_right:Nx #1#2 {
4169   \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 }

```

```

4170 }
4171 \cs_new_protected:Npn \tl_put_right:No #1#2 {
4172   \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} }
4173 }
4174 \cs_new_protected:Npn \tl_gput_right:Nn #1#2 {
4175   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} }
4176 }
4177 \cs_new_protected:Npn \tl_gput_right:NV #1#2 {
4178   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 }
4179 }
4180 \cs_new_protected:Npn \tl_gput_right:Nv #1#2 {
4181   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:v {#2} }
4182 }
4183 \cs_new_protected:Npn \tl_gput_right:No #1#2 {
4184   \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} }
4185 }
4186 \cs_new_protected:Npn \tl_gput_right:Nx #1#2 {
4187   \cs_gset_nopar:Npx #1 { \exp_not:o #1 #2 }
4188 }
4189 \cs_generate_variant:Nn \tl_put_right:Nn { c }
4190 \cs_generate_variant:Nn \tl_put_right:NV { c }
4191 \cs_generate_variant:Nn \tl_put_right:Nv { c }
4192 \cs_generate_variant:Nn \tl_put_right:Nx { c }
4193 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
4194 \cs_generate_variant:Nn \tl_gput_right:NV { c }
4195 \cs_generate_variant:Nn \tl_gput_right:Nv { c }
4196 \cs_generate_variant:Nn \tl_gput_right:No { c }
4197 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

(End definition for `\tl_put_right:Nn`. This function is documented on page ??.)

`\tl_put_left:Nn` Adding to the left is basically the same as putting on the right.

```

\tl_put_left:NV
\tl_put_left:Nv
\tl_put_left:No
\tl_put_left:Nx
\tl_put_left:cn
\tl_put_left:cV
\tl_put_left:cv
\tl_put_left:cx
\tl_gput_left:Nn
\tl_gput_left:NV
\tl_gput_left:Nv
\tl_gput_left:No
\tl_gput_left:Nx
\tl_gput_left:cn
\tl_gput_left:cV
\tl_gput_left:cv
\tl_gput_left:cx

```

```

4198 \cs_new_protected:Npn \tl_put_left:Nn #1#2 {
4199   \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 }
4200 }
4201 \cs_new_protected:Npn \tl_put_left:NV #1#2 {
4202   \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 }
4203 }
4204 \cs_new_protected:Npn \tl_put_left:Nv #1#2 {
4205   \cs_set_nopar:Npx #1 { \exp_not:v {#2} \exp_not:o #1 }
4206 }
4207 \cs_new_protected:Npn \tl_put_left:Nx #1#2 {
4208   \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 }
4209 }
4210 \cs_new_protected:Npn \tl_put_left:No #1#2 {
4211   \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 }
4212 }
4213 \cs_new_protected:Npn \tl_gput_left:Nn #1#2 {
4214   \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 }

```

```

4215 }
4216 \cs_new_protected:Npn \tl_gput_left:NV #1#2 {
4217   \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 }
4218 }
4219 \cs_new_protected:Npn \tl_gput_left:Nv #1#2 {
4220   \cs_gset_nopar:Npx #1 { \exp_not:v {#2} \exp_not:o #1 }
4221 }
4222 \cs_new_protected:Npn \tl_gput_left:No #1#2 {
4223   \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 }
4224 }
4225 \cs_new_protected:Npn \tl_gput_left:Nx #1#2 {
4226   \cs_gset_nopar:Npx #1 { #2 \exp_not:o #1 }
4227 }
4228 \cs_generate_variant:Nn \tl_put_left:Nn { c }
4229 \cs_generate_variant:Nn \tl_put_left:NV { c }
4230 \cs_generate_variant:Nn \tl_put_left:Nv { c }
4231 \cs_generate_variant:Nn \tl_put_left:Nx { c }
4232 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
4233 \cs_generate_variant:Nn \tl_gput_left:NV { c }
4234 \cs_generate_variant:Nn \tl_gput_left:Nv { c }
4235 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

(End definition for `\tl_put_left:Nn`. This function is documented on page ??.)

`\tl_gset:Nc` These two functions are included because they are necessary in Denys' implementations.
`\tl_set:Nc` The `:Nc` convention (see the expansion module) is very unusual at first sight, but it works nicely over all modules, so we would like to keep it.

Construct a control sequence on the fly from `#2` and save it in `#1`.

```

4236 \cs_new_protected_nopar:Npn \tl_gset:Nc {
4237   <*check>
4238   \pref_global_chk:
4239   </check>
4240   <-check> \pref_global:D
4241   \tl_set:Nc}

```

`\pref_global_chk:` will turn the variable check in `\tl_set:No` into a global check.

```

4242 \cs_new_protected_nopar:Npn \tl_set:Nc #1#2{\tl_set:No #1{\cs:w#2\cs_end:}}

```

(End definition for `\tl_gset:Nc`. This function is documented on page ??.)

102.2 Variables and constants

`\c_job_name_tl` Inherited from the `expl3` name for the primitive: this needs to actually contain the text of the jobname rather than the name of the primitive, of course.

```

4243 \tl_new:N \c_job_name_tl
4244 \tl_set:Nx \c_job_name_tl { \tex_jobname:D }

```

(End definition for `\c_job_name_tl`. This function is documented on page 83.)

`\c_empty_tl` Two constants which are often used.

```
4245 \tl_const:Nn \c_empty_tl { }
```

(End definition for `\c_empty_tl`. This function is documented on page 83.)

`\c_space_tl` A space as a token list (as opposed to as a character).

```
4246 \tl_const:Nn \c_space_tl { ~ }
```

(End definition for `\c_space_tl`. This function is documented on page 83.)

`\g_tmpa_tl` Global temporary token list variables. They are supposed to be set and used immediately,
`\g_tmpb_tl` with no delay between the definition and the use because you can't count on other macros
not to redefine them from under you.

```
4247 \tl_new:N \g_tmpa_tl
```

```
4248 \tl_new:N \g_tmpb_tl
```

(End definition for `\g_tmpa_tl`. This function is documented on page 83.)

`\l_kernel_testa_tl` Local temporaries. These are the ones for test routines. This means that one can safely
`\l_kernel_testb_tl` use other temporaries when calling test routines.

```
4249 \tl_new:N \l_kernel_testa_tl
```

```
4250 \tl_new:N \l_kernel_testb_tl
```

(End definition for `\l_kernel_testa_tl`. This function is documented on page 83.)

`\l_tmpa_tl` These are local temporary token list variables. Be sure not to assume that the value you
`\l_tmpb_tl` put into them will survive for long—see discussion above.

```
4251 \tl_new:N \l_tmpa_tl
```

```
4252 \tl_new:N \l_tmpb_tl
```

(End definition for `\l_tmpa_tl`. This function is documented on page 83.)

`\l_kernel_tmpa_tl` These are local temporary token list variables reserved for use by the kernel. They should
`\l_kernel_tmpb_tl` not be used by other modules.

```
4253 \tl_new:N \l_kernel_tmpa_tl
```

```
4254 \tl_new:N \l_kernel_tmpb_tl
```

(End definition for `\l_kernel_tmpa_tl`. This function is documented on page 83.)

102.3 Predicates and conditionals

We also provide a few conditionals, both in expandable form (with `\c_true_bool`) and in ‘brace-form’, the latter are denoted by TF at the end, as explained elsewhere.

`\tl_if_empty_p:N` These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```

4255 \prg_set_conditional:Npnn \tl_if_empty:N #1 {p,TF,T,F} {
4256   \if_meaning:w #1 \c_empty_tl
4257   \prg_return_true: \else: \prg_return_false: \fi:
4258 }
4259 \cs_generate_variant:Nn \tl_if_empty_p:N {c}
4260 \cs_generate_variant:Nn \tl_if_empty:N {NTF} {c}
4261 \cs_generate_variant:Nn \tl_if_empty:NT {c}
4262 \cs_generate_variant:Nn \tl_if_empty:NF {c}

```

(End definition for `\tl_if_empty_p:N` and `\tl_if_empty_p:c`. These functions are documented on page 80.)

`\tl_if_eq_p:NN` Returns `\c_true_bool` iff the two token list variables are equal.

```

4263 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 {p,TF,T,F} {
4264   \if_meaning:w #1 #2 \prg_return_true: \else: \prg_return_false: \fi:
4265 }
4266 \cs_generate_variant:Nn \tl_if_eq_p:NN {Nc,c,cc}
4267 \cs_generate_variant:Nn \tl_if_eq:NNTF {Nc,c,cc}
4268 \cs_generate_variant:Nn \tl_if_eq:NNT {Nc,c,cc}
4269 \cs_generate_variant:Nn \tl_if_eq:NNF {Nc,c,cc}

```

(End definition for `\tl_if_eq_p:NN` and others. These functions are documented on page 80.)

`\tl_if_eq:nnTF` A simple store and compare routine.

```

\l_tl_tmpa_tl
\l_tl_tmpp_tl
4270 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF } {
4271   \group_begin:
4272     \tl_set:Nn \l_tl_tmpa_tl {#1}
4273     \tl_set:Nn \l_tl_tmpp_tl {#2}
4274     \tex_ifx:D \l_tl_tmpa_tl \l_tl_tmpp_tl
4275     \group_end:
4276     \prg_return_true:
4277     \tex_else:D
4278     \group_end:
4279     \prg_return_false:
4280     \tex_fi:D
4281 }
4282 \tl_new:N \l_tl_tmpa_tl
4283 \tl_new:N \l_tl_tmpp_tl

```

(End definition for `\tl_if_eq:nn`. This function is documented on page 81.)

`\tl_if_empty_p:n` It would be tempting to just use `\if_meaning:w\q_nil#1\q_nil` as a test since this works really well. However it fails on a token list starting with `\q_nil` of course but more troubling is the case where argument is a complete conditional such as `\if_true: a \else: b \fi:` because then `\if_true:` is used by `\if_meaning:w`, the test turns out false, the `\else:` executes the false branch, the `\fi:` ends it and the `\q_nil` at the end starts executing... A safer route is to convert the entire token list into harmless characters first and then compare that. This way the test will even accept `\q_nil` as the first token.

```

4284 \prg_new_conditional:Npnn \tl_if_empty:n #1 {p,TF,T,F} {
4285   \exp_after:wN \if_meaning:w \exp_after:wN \q_nil \tl_to_str:n {#1} \q_nil
4286   \prg_return_true: \else: \prg_return_false: \fi:
4287 }
4288 \cs_generate_variant:Nn \tl_if_empty_p:n {V}
4289 \cs_generate_variant:Nn \tl_if_empty:nTF {V}
4290 \cs_generate_variant:Nn \tl_if_empty:nT {V}
4291 \cs_generate_variant:Nn \tl_if_empty:nF {V}
4292 \cs_generate_variant:Nn \tl_if_empty_p:n {o}
4293 \cs_generate_variant:Nn \tl_if_empty:nTF {o}
4294 \cs_generate_variant:Nn \tl_if_empty:nT {o}
4295 \cs_generate_variant:Nn \tl_if_empty:nF {o}

```

(End definition for `\tl_if_empty_p:n`, `\tl_if_empty_p:V`, and `\tl_if_empty_p:o`. These functions are documented on page 80.)

`\tl_if_blank_p:n` This is based on the answers in “Around the Bend No 2” but is safer as the tests listed there all have one small flaw: If the input in the test is two tokens with the same meaning as the internal delimiter, they will fail since one of them is mistaken for the actual delimiter. In our version below we make sure to pass the input through `\tl_to_str:n` which ensures that all the tokens are converted to catcode 12. However we use an `a` with catcode 11 as delimiter so we can *never* get into the same problem as the solutions in “Around the Bend No 2”.

```

4296 \prg_new_conditional:Npnn \tl_if_blank:n #1 {p,TF,T,F} {
4297   \exp_after:wN \tl_if_blank_p_aux:w \tl_to_str:n {#1} aa.\q_stop
4298 }
4299 \cs_new:Npn \tl_if_blank_p_aux:w #1#2 a #3#4 \q_stop {
4300   \if_meaning:w #3 #4 \prg_return_true: \else: \prg_return_false: \fi:
4301 }
4302 \cs_generate_variant:Nn \tl_if_blank_p:n {V}
4303 \cs_generate_variant:Nn \tl_if_blank:nTF {V}
4304 \cs_generate_variant:Nn \tl_if_blank:nT {V}
4305 \cs_generate_variant:Nn \tl_if_blank:nF {V}
4306 \cs_generate_variant:Nn \tl_if_blank_p:n {o}
4307 \cs_generate_variant:Nn \tl_if_blank:nTF {o}
4308 \cs_generate_variant:Nn \tl_if_blank:nT {o}
4309 \cs_generate_variant:Nn \tl_if_blank:nF {o}

```

(End definition for `\tl_if_blank_p:n`, `\tl_if_blank_p:V`, and `\tl_if_blank_p:o`. These functions are documented on page 81.)

`\tl_if_single:nTF` If the argument is a single token. ‘Space’ is considered ‘true’.
`\tl_if_single_p:n`

```

4310 \prg_new_conditional:Nnn \tl_if_single:n {p,TF,T,F} {
4311   \tl_if_empty:nTF {#1}
4312     {\prg_return_false:}
4313     {
4314       \tl_if_blank:nTF {#1}
4315         {\prg_return_true:}
4316         {
4317           \tl_if_single_aux:w #1 \q_stop
4318         }
4319     }
4320 }
```

Use `\exp_after:wN` below I know what I’m doing. Use `\exp_args:NV` or `\exp_args_unbraced:NV` for more flexibility in your own code.

```

4321 \prg_new_conditional:Nnn \tl_if_single:N {p,TF,T,F} {
4322   \tl_if_empty:NTF #1
4323     { \prg_return_false: }
4324     {
4325       \tl_if_blank:oTF {#1}
4326       { \prg_return_true: }
4327       { \exp_after:wN \tl_if_single_aux:w #1 \q_stop }
4328     }
4329 }

4330 \cs_new:Npn \tl_if_single_aux:w #1#2 \q_stop {
4331   \tl_if_empty:nTF {#2} \prg_return_true: \prg_return_false:
4332 }
```

(End definition for `\tl_if_single:n`. This function is documented on page 81.)

102.4 Working with the contents of token lists

`\tl_to_lowercase:n` Just some names for a few primitives.
`\tl_to_uppercase:n`

```

4333 \cs_new_eq:NN \tl_to_lowercase:n \tex_lowercase:D
4334 \cs_new_eq:NN \tl_to_uppercase:n \tex_uppercase:D
```

(End definition for `\tl_to_lowercase:n`. This function is documented on page 81.)

`\tl_to_str:n` Another name for a primitive.

```

4335 \cs_new_eq:NN \tl_to_str:n \etex_detokenize:D
```

(End definition for `\tl_to_str:n`. This function is documented on page 79.)

`\tl_to_str:N` These functions return the replacement text of a token list as a string list with all characters catcoded to ‘other’.

`\tl_to_str:c`

`\tl_to_str_aux:w`

```

4336 \cs_new_nopar:Npn \tl_to_str:N {\exp_after:wN\tl_to_str_aux:w
4337   \token_to_meaning:N}
4338 \cs_new_nopar:Npn \tl_to_str_aux:w #1>{}
4339 \cs_generate_variant:Nn \tl_to_str:N {c}

```

(End definition for `\tl_to_str:N`. This function is documented on page 79.)

`\tl_map_function:nN` Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker will be read immediately and the loop terminated.

`\tl_map_function:NN`

`\tl_map_function:cN`

`\tl_map_function_aux:NN`

```

4340 \cs_new:Npn \tl_map_function:nN #1#2{
4341   \tl_map_function_aux:Nn #2 #1 \q_recursion_tail \q_recursion_stop
4342 }
4343 \cs_new_nopar:Npn \tl_map_function:NN #1#2{
4344   \exp_after:wN \tl_map_function_aux:Nn
4345   \exp_after:wN #2 #1 \q_recursion_tail \q_recursion_stop
4346 }
4347 \cs_new:Npn \tl_map_function_aux:NN #1#2{
4348   \quark_if_recursion_tail_stop:n{#2}
4349   #1{#2} \tl_map_function_aux:Nn #1
4350 }
4351 \cs_generate_variant:Nn \tl_map_function:NN {cN}

```

(End definition for `\tl_map_function:nN`. This function is documented on page 81.)

`\tl_map_inline:nn` The inline functions are straight forward by now. We use a little trick with the counter `\g_tl_inline_level_int` to make them nestable. We can also make use of `\tl_map_function:Nn` from before.

`\tl_map_inline:Nn`

`\tl_map_inline:cn`

`\tl_map_inline_aux:n`

`\g_tl_inline_level_int`

```

4352 \cs_new_protected:Npn \tl_map_inline:nn #1#2{
4353   \int_gincr:N \g_tl_inline_level_int
4354   \cs_gset:cpn {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
4355   ##1{#2}
4356   \exp_args:Nc \tl_map_function_aux:Nn
4357   {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
4358   #1 \q_recursion_tail\q_recursion_stop
4359   \int_gdecr:N \g_tl_inline_level_int
4360 }
4361 \cs_new_protected:Npn \tl_map_inline:Nn #1#2{
4362   \int_gincr:N \g_tl_inline_level_int
4363   \cs_gset:cpn {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
4364   ##1{#2}
4365   \exp_last_unbraced:NcV \tl_map_function_aux:Nn
4366   {tl_map_inline_ \int_use:N \g_tl_inline_level_int :n}
4367   #1 \q_recursion_tail\q_recursion_stop
4368   \int_gdecr:N \g_tl_inline_level_int

```

```

4369 }
4370 \cs_generate_variant:Nn \tl_map_inline:Nn {c}

```

(End definition for `\tl_map_inline:nn`. This function is documented on page 83.)

`\tl_map_variable:nNn` `\tl_map_variable:NNn` `\tl_map_variable:cNn` `\tl_map_variable:nNn` $\langle token\ list \rangle$ $\langle temp \rangle$ $\langle action \rangle$ assigns $\langle temp \rangle$ to each element and executes $\langle action \rangle$.

```

4371 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3{
4372   \tl_map_variable_aux:Nnn #2 {#3} #1 \q_recursion_tail \q_recursion_stop
4373 }

```

Next really has to be v/V args

```

4374 \cs_new_protected_nopar:Npn \tl_map_variable:NNn {\exp_args:No \tl_map_variable:nNn}
4375 \cs_generate_variant:Nn \tl_map_variable:NNn {c}

```

(End definition for `\tl_map_variable:nNn`. This function is documented on page 82.)

`\tl_map_variable_aux:NnN` The general loop. Assign the temp variable #1 to the current item #3 and then check if that's the stop marker. If it is, break the loop. If not, execute the action #2 and continue.

```

4376 \cs_new_protected:Npn \tl_map_variable_aux:Nnn #1#2#3{
4377   \tl_set:Nn #1{#3}
4378   \quark_if_recursion_tail_stop:N #1
4379   #2 \tl_map_variable_aux:Nnn #1{#2}
4380 }

```

(End definition for `\tl_map_variable_aux:NnN`.)

`\tl_map_break:` The break statement.

```

4381 \cs_new_eq:NN \tl_map_break: \use_none_delimit_by_q_recursion_stop:w

```

(End definition for `\tl_map_break:.` This function is documented on page 82.)

`\tl_reverse:n` `\tl_reverse:V` `\tl_reverse:o` Reversal of a token list is done by taking one token at a time and putting it in front of the ones before it.

```

\tl_reverse_aux:nN 4382 \cs_new:Npn \tl_reverse:n #1{
4383   \tl_reverse_aux:nN {} #1 \q_recursion_tail\q_recursion_stop
4384 }
4385 \cs_new:Npn \tl_reverse_aux:nN #1#2{
4386   \quark_if_recursion_tail_stop_do:nn {#2}{ #1 }
4387   \tl_reverse_aux:nN {#2#1}
4388 }
4389 \cs_generate_variant:Nn \tl_reverse:n {V,o}

```

(End definition for `\tl_reverse:n`. This function is documented on page 82.)

\tl_reverse:N This reverses the list, leaving \exp_stop_f: in front, which in turn is removed by the f expansion which comes to a halt.

```

4390 \cs_new_protected_nopar:Npn \tl_reverse:N #1 {
4391   \tl_set:Nf #1 { \tl_reverse:o { #1 \exp_stop_f: } }
4392 }

```

(End definition for \tl_reverse:N. This function is documented on page 82.)

\tl_elt_count:n Count number of elements within a token list or token list variable. Brace groups within
\tl_elt_count:V the list are read as a single element. \tl_elt_count_aux:n grabs the element and
\tl_elt_count:o replaces it by +1. The 0 to ensure it works on an empty list.
\tl_elt_count:N

```

4393 \cs_new:Npn \tl_elt_count:n #1{
4394   \int_eval:n {
4395     0 \tl_map_function:nN {#1} \tl_elt_count_aux:n
4396   }
4397 }
4398 \cs_generate_variant:Nn \tl_elt_count:n {V,o}
4399 \cs_new_nopar:Npn \tl_elt_count:N #1{
4400   \int_eval:n {
4401     0 \tl_map_function:NN #1 \tl_elt_count_aux:n
4402   }
4403 }

```

(End definition for \tl_elt_count:n. This function is documented on page 82.)

\tl_num_elt_count_aux:n Helper function for counting elements in a token list.

```

4404 \cs_new:Npn \tl_num_elt_count_aux:n #1 { + 1 }

```

(End definition for \tl_num_elt_count_aux:n.)

\tl_set_rescan:Nnn These functions store the {<token list>} in <tl var.> after redefining catcodes, etc., in
\tl_gset_rescan:Nnn argument #2.

\tl_set_rescan:Nno #1 : <tl var.>
\tl_gset_rescan:Nno #2 : {<catcode setup, etc.>}
 #3 : {<token list>}

```

4405 \cs_new_protected:Npn \tl_set_rescan:Nnn { \tl_set_rescan_aux:NNnn \tl_set:Nn }
4406 \cs_new_protected:Npn \tl_gset_rescan:Nnn { \tl_set_rescan_aux:NNnn \tl_gset:Nn }
4407 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno }
4408 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno }

```

\tl_set_rescan_aux:NNnn This macro uses a trick to extract an unexpanded token list after it's rescanned with \etex_scantokens:D. This technique was first used (as far as I know) by Heiko Oberdiek in his catchfile package, albeit for real files rather than the 'fake' \scantokens one.

The basic problem arises because \etex_scantokens:D emulates a file read, which inserts an EOF marker into the expansion; the simplistic

`\exp_args:NNo \cs_set:Npn \tmp:w { \etex_scantokens:D {some text} }`
 unfortunately doesn't work, calling the error:
! File ended while scanning definition of \tmp:w.
 (LuaTeX works around this problem with its `\scantextokens` primitive.)

Usually, we'd define `\etex_everyeof:D` to be `\exp_not:N` to gobble the EOF marker, but since we're not expanding the token list, it gets left in there and we have the same basic problem.

Instead, we define `\etex_everyeof:D` to contain a marker that's impossible to occur within the scanned text; that is, the same char twice with different catcodes. (For some reason, we *don't* need to insert a `\exp_not:N` token after it to prevent the EOF marker to expand. Anyone know why?)

A helper function is can be used to save the token list delimited by the special marker, keeping the catcode redefinitions hidden away in a group.

`\c_two_ats_with_two_catcodes_tl` A tl with two @ characters with two different catcodes. Used as a special marker for delimited text.

```

4409 \group_begin:
4410 \tex_lccode:D '\A = '\@ \scan_stop:
4411 \tex_lccode:D '\B = '\@ \scan_stop:
4412 \tex_catcode:D '\A = 8 \scan_stop:
4413 \tex_catcode:D '\B = 3 \scan_stop:
4414 \tl_to_lowercase:n {
4415   \group_end:
4416   \tl_const:Nn \c_two_ats_with_two_catcodes_tl { A B }
4417 }

```

#1 : `\tl_set` function
 #2 : `\tl var.`
 #3 : `{\catcode setup, etc.}`
 #4 : `{\token list}`

Note that if you change `\etex_everyeof:D` in #3 then you'd better do it correctly!

```

4418 \cs_new_protected:Npn \tl_set_rescan_aux:NNnn #1#2#3#4 {
4419   \group_begin:
4420     \toks_set:NV \etex_everyeof:D \c_two_ats_with_two_catcodes_tl
4421     \tex_endlinechar:D = \c_minus_one
4422     #3
4423     \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#4}
4424     \exp_args:NNNV
4425   \group_end:
4426   #1 #2 \l_tmpa_tl
4427 }

```

`\tl_rescan_aux:w`

```

4428 \exp_after:wN \cs_set:Npn
4429 \exp_after:wN \tl_rescan_aux:w
4430 \exp_after:wN #
4431 \exp_after:wN 1 \c_two_ats_with_two_catcodes_tl {
4432   \tl_set:Nn \l_tmpa_tl {#1}
4433 }

```

(End definition for `\tl_set_rescan:Nnn` and `\tl_gset_rescan:Nnn`. These functions are documented on page 80.)

`\tl_set_rescan:Nnx` `\tl_gset_rescan:Nnx` These functions store the full expansion of $\{\langle token\ list\rangle\}$ in $\langle tl\ var.\rangle$ after redefining catcodes, etc., in argument #2.

```

#1 :  $\langle tl\ var.\rangle$ 
#2 :  $\{\langle catcode\ setup,\ etc.\rangle\}$ 
#3 :  $\{\langle token\ list\rangle\}$ 

```

The expanded versions are much simpler because the `\etex_scantokens:D` can occur within the expansion.

```

4434 \cs_new_protected:Npn \tl_set_rescan:Nnx #1#2#3 {
4435   \group_begin:
4436     \etex_veryeof:D { \exp_not:N }
4437     \tex_endlinechar:D = \c_minus_one
4438     #2
4439     \tl_set:Nx \l_kernel_tmpa_tl { \etex_scantokens:D {#3} }
4440     \exp_args:NNNV
4441   \group_end:
4442   \tl_set:Nn #1 \l_kernel_tmpa_tl
4443 }

```

Globally is easier again:

```

4444 \cs_new_protected:Npn \tl_gset_rescan:Nnx #1#2#3 {
4445   \group_begin:
4446     \etex_veryeof:D { \exp_not:N }
4447     \tex_endlinechar:D = \c_minus_one
4448     #2
4449     \tl_gset:Nx #1 { \etex_scantokens:D {#3} }
4450   \group_end:
4451 }

```

(End definition for `\tl_set_rescan:Nnx` and `\tl_gset_rescan:Nnx`. These functions are documented on page 80.)

`\tl_rescan:nn` The inline wrapper for `\etex_scantokens:D`.

```

#1 : Catcode changes (etc.)
#2 : Token list to re-tokenise

```



```

4452 \cs_new_protected:Npn \tl_rescan:nn #1#2 {
4453   \group_begin:
4454     \toks_set:NV \etex_everyeof:D \c_two_at_with_two_catcodes_tl
4455     \tex_endlinechar:D = \c_minus_one
4456     #1
4457     \exp_after:wN \tl_rescan_aux:w \etex_scantokens:D {#2}
4458   \exp_args:NV \group_end:
4459   \l_tmpa_tl
4460 }

```

(End definition for `\tl_rescan:nn`. This function is documented on page 79.)

102.5 Checking for and replacing tokens

`\tl_if_in:NnTF` See the replace functions for further comments. In this part we don't care too much about brace stripping since we are not interested in passing on the tokens which are split off in the process.

`\tl_if_in:cnTF`

```

4461 \prg_new_protected_conditional:Npnn \tl_if_in:Nn #1#2 {TF,T,F} {
4462   \cs_set:Npn \tl_tmp:w ##1 #2 ##2 \q_stop {
4463     \quark_if_no_value:nTF {##2} {\prg_return_false:} {\prg_return_true:}
4464   }
4465   \exp_after:wN \tl_tmp:w #1 #2 \q_no_value \q_stop
4466 }
4467 \cs_generate_variant:Nn \tl_if_in:NnTF {c}
4468 \cs_generate_variant:Nn \tl_if_in:NnT {c}
4469 \cs_generate_variant:Nn \tl_if_in:NnF {c}

```

(End definition for `\tl_if_in:Nn` and `\tl_if_in:cn`. These functions are documented on page 84.)

`\tl_if_in:nnTF`

`\tl_if_in:VnTF`

`\tl_if_in:onTF`

```

4470 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 {TF,T,F} {
4471   \cs_set:Npn \tl_tmp:w ##1 #2 ##2 \q_stop {
4472     \quark_if_no_value:nTF {##2} {\prg_return_false:} {\prg_return_true:}
4473   }
4474   \tl_tmp:w #1 #2 \q_no_value \q_stop
4475 }
4476 \cs_generate_variant:Nn \tl_if_in:nnTF {V}
4477 \cs_generate_variant:Nn \tl_if_in:nnT {V}
4478 \cs_generate_variant:Nn \tl_if_in:nnF {V}
4479 \cs_generate_variant:Nn \tl_if_in:nnTF {o}
4480 \cs_generate_variant:Nn \tl_if_in:nnT {o}
4481 \cs_generate_variant:Nn \tl_if_in:nnF {o}

```

(End definition for `\tl_if_in:nn`, `\tl_if_in:Vn`, and `\tl_if_in:on`. These functions are documented on page 84.)

`_l_tl_replace_tl`

`\tl_replace_in:Nnn`

`\tl_replace_in:cn`

`\tl_greplace_in:Nnn`

`\tl_greplace_in:cn`

`_tl_replace_in_aux:NNnn`

The concept here is that only the first occurrence should be replaced. The first step is to define an auxiliary which will match the appropriate item, with a trailing marker. If the

last token is the marker there is nothing to do, otherwise replace the token and clean up (hence the second use of `_tl_tmp:w`). To prevent losing braces or spaces there are a couple of empty groups and the strange-looking `\use:n`.

```

4482 \tl_new:N \_l\_tl\_replace\_tl
4483 \cs_new_protected_nopar:Npn \tl\_replace\_in:Nnn {
4484   \_tl\_replace\_in\_aux:NNnn \tl\_set\_eq:NN
4485 }
4486 \cs_new_protected:Npn \tl\_replace\_in\_aux:NNnn #1#2#3#4 {
4487   \cs\_set\_protected:Npn \_tl\_tmp:w ##1 #3 ##2 \q\_stop
4488   {
4489     \quark\_if\_no\_value:nF {##2}
4490     {
4491       \tl\_set:No \_l\_tl\_replace\_tl { ##1 #4 }
4492       \cs\_set\_protected:Npn \_tl\_tmp:w #####1 \q\_nil #3 \q\_no\_value
4493       { \tl\_put\_right:No \_l\_tl\_replace\_tl {#####1} }
4494       \_tl\_tmp:w \prg\_do\_nothing: ##2
4495       #1 #2 \_l\_tl\_replace\_tl
4496     }
4497   }
4498   \use:n
4499   {
4500     \exp\_after:wN \_tl\_tmp:w \exp\_after:wN
4501     \prg\_do\_nothing:
4502   }
4503   #2 \q\_nil #3 \q\_no\_value \q\_stop
4504 }
4505 \cs_new_protected_nopar:Npn \tl\_greplace\_in:Nnn {
4506   \_tl\_replace\_in\_aux:NNnn \tl\_gset\_eq:NN
4507 }
4508 \cs_generate_variant:Nn \tl\_replace\_in:Nnn { c }
4509 \cs_generate_variant:Nn \tl\_greplace\_in:Nnn { c }

```

(End definition for `_l_tl_replace_tl`. This function is documented on page 84.)

```

\tl\_replace\_all\_in:Nnn
\tl\_replace\_all\_in:Nnn
\tl\_greplace\_all\_in:cn
\tl\_greplace\_all\_in:cn
\_tl\_replace\_all\_in\_aux:NNnn

```

A similar approach here but with a loop built in.

```

4510 \cs_new_protected_nopar:Npn \tl\_replace\_all\_in:Nnn {
4511   \_tl\_replace\_all\_in\_aux:NNnn \tl\_set\_eq:NN
4512 }
4513 \cs_new_protected_nopar:Npn \tl\_greplace\_all\_in:Nnn {
4514   \_tl\_replace\_all\_in\_aux:NNnn \tl\_gset\_eq:NN
4515 }
4516 \cs_new_protected:Npn \tl\_replace\_all\_in\_aux:NNnn #1#2#3#4 {
4517   \tl\_clear:N \_l\_tl\_replace\_tl
4518   \cs\_set\_protected:Npn \_tl\_tmp:w ##1 #3 ##2 \q\_stop
4519   {
4520     \quark\_if\_no\_value:nTF {##2}
4521     {
4522       \cs\_set\_protected:Npn \_tl\_tmp:w #####1 \q\_nil #####2 \q\_stop

```

```

4523         { \tl_put_right:No \_l_tl_replace_tl {####1} }
4524         \_tl_tmp:w ##1 \q_stop
4525     }
4526     {
4527         \tl_put_right:No \_l_tl_replace_tl { ##1 #4 }
4528         \_tl_tmp:w \prg_do_nothing: ##2 \q_stop
4529     }
4530 }
4531 \use:n
4532 {
4533     \exp_after:wN \_tl_tmp:w \exp_after:wN
4534     \prg_do_nothing:
4535 }
4536 #2 \q_nil #3 \q_no_value \q_stop
4537 #1 #2 \_l_tl_replace_tl
4538 }
4539 \cs_generate_variant:Nn \tl_replace_all_in:Nnn { c }
4540 \cs_generate_variant:Nn \tl_greplace_all_in:Nnn { c }

```

(End definition for `\tl_replace_all_in:Nnn`. This function is documented on page 84.)

`\tl_remove_in:Nn` Next comes a series of removal functions. I have just implemented them as subcases of
`\tl_remove_in:cn` the replace functions for now (I'm lazy).

```

\tl_gremove_in:Nn
\tl_gremove_in:cn
4541 \cs_new_protected:Npn \tl_remove_in:Nn #1#2{\tl_replace_in:Nnn #1{#2}{}}
4542 \cs_new_protected:Npn \tl_gremove_in:Nn #1#2{\tl_greplace_in:Nnn #1{#2}{}}
4543 \cs_generate_variant:Nn \tl_remove_in:Nn {cn}
4544 \cs_generate_variant:Nn \tl_gremove_in:Nn {cn}

```

(End definition for `\tl_remove_in:Nn`. This function is documented on page 84.)

`\tl_remove_all_in:Nn` Same old, same old.

```

\tl_remove_all_in:cn
\tl_gremove_all_in:Nn
\tl_gremove_all_in:cn
4545 \cs_new_protected:Npn \tl_remove_all_in:Nn #1#2{
4546     \tl_replace_all_in:Nnn #1{#2}{}}
4547 }
4548 \cs_new_protected:Npn \tl_gremove_all_in:Nn #1#2{
4549     \tl_greplace_all_in:Nnn #1{#2}{}}
4550 }
4551 \cs_generate_variant:Nn \tl_remove_all_in:Nn {cn}
4552 \cs_generate_variant:Nn \tl_gremove_all_in:Nn {cn}

```

(End definition for `\tl_remove_all_in:Nn`. This function is documented on page 84.)

102.6 Heads or tails?

`\tl_head:n` These functions pick up either the head or the tail of a list. `\tl_head_iii:n` returns the
`\tl_head:V` first three items on a list.
`\tl_head:v`

`\tl_head_i:n`
`\tl_tail:n`
`\tl_tail:V`
`\tl_tail:v`
`\tl_tail:f`
`\tl_head_iii:n`
`\tl_head_iii:f`
`\tl_head:w`
`\tl_head_i:w`
`\tl_tail:w`

```

4553 \cs_new:Npn \tl_head:n #1{\tl_head:w #1\q_stop}
4554 \cs_new_eq:NN \tl_head_i:n \tl_head:n
4555 \cs_new:Npn \tl_tail:n #1{\tl_tail:w #1\q_stop}
4556 \cs_generate_variant:Nn \tl_tail:n {f}
4557 \cs_new:Npn \tl_head_iii:n #1{\tl_head_iii:w #1\q_stop}
4558 \cs_generate_variant:Nn \tl_head_iii:n {f}
4559 \cs_new:Npn \tl_head:w #1#2\q_stop{#1}
4560 \cs_new_eq:NN \tl_head_i:w \tl_head:w
4561 \cs_new:Npn \tl_tail:w #1#2\q_stop{#2}
4562 \cs_new:Npn \tl_head_iii:w #1#2#3#4\q_stop{#1#2#3}
4563 \cs_generate_variant:Nn \tl_head:n { V }
4564 \cs_generate_variant:Nn \tl_head:n { v }
4565 \cs_generate_variant:Nn \tl_tail:n { V }
4566 \cs_generate_variant:Nn \tl_tail:n { v }

```

(End definition for `\tl_head:n`. This function is documented on page 85.)

`\tl_if_head_eq_meaning_p:nN` When we want to check if the first token of a list equals something specific it is usually either to see if it is a control sequence or a character. Hence we make two different functions as the internal test is different. `\tl_if_head_meaning_eq:nNTF` uses `\if_meaning:w` and will consider the tokens `b11` and `b12` different. `\tl_if_head_char_eq:nNTF` on the other hand only compares character codes so would regard `b11` and `b12` as equal but would also regard two primitives as equal.

```

4567 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 {p,TF,T,F} {
4568   \exp_after:wN \if_meaning:w \tl_head:w #1 \q_stop #2
4569   \prg_return_true: \else: \prg_return_false: \fi:
4570 }

```

For the charcode and catcode versions we insert `\exp_not:N` in front of both tokens. If you need them to expand fully as `TEX` does itself with these you can use an `f` type expansion.

```

4571 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 {p,TF,T,F} {
4572   \exp_after:wN \if:w \exp_after:wN \exp_not:N
4573   \tl_head:w #1 \q_stop \exp_not:N #2
4574   \prg_return_true: \else: \prg_return_false: \fi:
4575 }

```

Actually the default is already an `f` type expansion.

```

4576 %% \cs_new:Npn \tl_if_head_eq_charcode_p:fN #1#2{
4577 %%   \exp_after:wN \if_charcode:w \tl_head:w #1\q_stop\exp_not:N#2
4578 %%   \c_true_bool
4579 %%   \else:
4580 %%   \c_false_bool
4581 %%   \fi:
4582 %% }
4583 %% \def_long_test_function_new:npn {tl_if_head_eq_charcode:fN}#1#2{
4584 %%   \if_predicate:w \tl_if_head_eq_charcode_p:fN {#1}#2}

```

These :fN variants are broken; temporary patch:

```

4585 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN {f}
4586 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF {f}
4587 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT {f}
4588 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF {f}

```

And now catcodes:

```

4589 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1#2 {p,TF,T,F} {
4590   \exp_after:wN \if_catcode:w \exp_after:wN \exp_not:N
4591     \tl_head:w #1 \q_stop \exp_not:N #2
4592   \prg_return_true: \else: \prg_return_false: \fi:
4593 }

```

(End definition for `\tl_if_head_eq_meaning_p:nN`. This function is documented on page 86.)

103 l3toks implementation

We start by ensuring that the required packages are loaded.

```

4594 <*package>
4595 \ProvidesExplPackage
4596   {\filename}{\filedate}{\fileversion}{\filedescription}
4597 \package_check_loaded_expl:
4598 </package>
4599 <*initex | package>

```

103.1 Allocation and use

`\toks_new:N` Allocates a new token register.
`\toks_new:c`

```

4600 <*initex>
4601 \alloc_new:nnnN {toks} \c_zero \c_max_register_int \tex_toksdef:D
4602 </initex>

4603 <*package>
4604 \cs_new_protected_nopar:Npn \toks_new:N #1 {
4605   \chk_if_free_cs:N #1
4606   \newtoks #1
4607 }
4608 </package>

4609 \cs_generate_variant:Nn \toks_new:N {c}

```

(End definition for `\toks_new:N` and `\toks_new:c`. These functions are documented on page 86.)

`\toks_use:N` This function returns the contents of a token register.

`\toks_use:c`

```
4610 \cs_new_eq:NN \toks_use:N \tex_the:D
4611 \cs_generate_variant:Nn \toks_use:N {c}
```

(End definition for `\toks_use:N`. This function is documented on page 87.)

`\toks_set:Nn` `\toks_set:Nn<toks><stuff>` stores `<stuff>` without expansion in `<toks>`. `\toks_set:No` and `\toks_set:Nx` expand `<stuff>` once and fully.

`\toks_set:Nv`

`\toks_set:No`

`\toks_set:Nx`

`\toks_set:Nf`

`\toks_set:cn`

`\toks_set:co`

`\toks_set:cV`

`\toks_set:cv`

`\toks_set:cx`

`\toks_set:cf`

```
4612 <*check>
4613 \cs_new_protected_nopar:Npn \toks_set:Nn #1 { \chk_local:N #1 #1 }
4614 \cs_generate_variant:Nn \toks_set:Nn {No,Nf}
4615 </check>
```

If we don't check if `<toks>` is a local register then the `\toks_set:Nn` function has nothing to do. We implement `\toks_set:No/d/f` by hand when not checking because this is going to be used *extensively* in keyval processing! TODO: (Will) Can we get some numbers published on how necessary this is? On the other hand I'm happy to believe Morten :)

```
4616 <!*check>
4617 \cs_new_eq:NN \toks_set:Nn \prg_do_nothing:
4618 \cs_new_protected:Npn \toks_set:Nv #1#2 {
4619   #1 \exp_after:wN { \int_to_roman:w -'0 \exp_eval_register:N #2 }
4620 }
4621 \cs_new_protected:Npn \toks_set:Nv #1#2 {
4622   #1 \exp_after:wN { \int_to_roman:w -'0 \exp_eval_register:c {#2} }
4623 }
4624 \cs_new_protected:Npn \toks_set:No #1#2 { #1 \exp_after:wN {#2} }
4625 \cs_new_protected:Npn \toks_set:Nf #1#2 {
4626   #1 \exp_after:wN { \int_to_roman:w -'0#2 }
4627 }
4628 </!check>
```

```
4629 \cs_generate_variant:Nn \toks_set:Nn {Nx,cn,cV,cv,co,cx,cf}
```

(End definition for `\toks_set:Nn`. This function is documented on page 87.)

`\toks_gset:Nn` These functions are the global variants of the above.

`\toks_gset:Nv`

`\toks_gset:No`

`\toks_gset:Nx`

`\toks_gset:cn`

`\toks_gset:cV`

`\toks_gset:co`

`\toks_gset:cv`

`\toks_gset:cx`

`\toks_gset:cf`

```
4630 <check>\cs_new_protected_nopar:Npn \toks_gset:Nn #1 { \chk_global:N #1 \pref_global:D #1 }
4631 <!check>\cs_new_eq:NN \toks_gset:Nn \pref_global:D
4632 \cs_generate_variant:Nn \toks_gset:Nn {NV,No,Nx,cn,cV,co,cx}
```

(End definition for `\toks_gset:Nn`. This function is documented on page 87.)

`\toks_set_eq:NN` `\toks_set_eq:NN<toks1><toks2>` copies the contents of `<toks2>` in `<toks1>`.

`\toks_set_eq:Nc`

`\toks_set_eq:cN`

`\toks_set_eq:cc`

`\toks_gset_eq:NN`

`\toks_gset_eq:Nc`

`\toks_gset_eq:cN`

`\toks_gset_eq:cc`

```
4633 <*check>
4634 \cs_new_protected_nopar:Npn \toks_set_eq:NN #1#2 {
```

```

4635 \chk_local:N #1
4636 \chk_var_or_const:N #2
4637 #1 #2
4638 }
4639 \cs_new_protected_nopar:Npn\toks_gset_eq:NN #1#2 {
4640 \chk_global:N #1
4641 \chk_var_or_const:N #2
4642 \pref_global:D #1 #2
4643 }
4644 </check>
4645 <!*check>
4646 \cs_new_eq:NN \toks_set_eq:NN \prg_do_nothing:
4647 \cs_new_eq:NN \toks_gset_eq:NN \pref_global:D
4648 </!check>
4649 \cs_generate_variant:Nn \toks_set_eq:NN {Nc,cN,cc}
4650 \cs_generate_variant:Nn \toks_gset_eq:NN {Nc,cN,cc}

```

(End definition for \toks_set_eq:NN. This function is documented on page 88.)

\toks_clear:N These functions clear a token register, either locally or globally.

```

\toks_gclear:N
\toks_clear:c
\toks_gclear:c
4651 \cs_new_protected_nopar:Npn \toks_clear:N #1 {
4652 #1\c_empty_toks
4653 <check>\chk_local_or_pref_global:N #1
4654 }
4655 \cs_new_protected_nopar:Npn \toks_gclear:N {
4656 <check> \pref_global_chk:
4657 <!check> \pref_global:D
4658 \toks_clear:N
4659 }
4660 \cs_generate_variant:Nn \toks_clear:N {c}
4661 \cs_generate_variant:Nn \toks_gclear:N {c}

```

(End definition for \toks_clear:N and others. These functions are documented on page 88.)

\toks_use_clear:N These functions clear a token register (locally or globally) after returning the contents.

\toks_use_clear:c They make sure that clearing the register does not interfere with following tokens. In other words, the contents of the register might operate on what follows in the input stream.

\toks_use_gclear:N

\toks_use_gclear:c

```

4662 \cs_new_protected_nopar:Npn \toks_use_clear:N #1 {
4663 \exp_last_unbraced:NNV \toks_clear:N #1 #1
4664 }
4665 \cs_new_protected_nopar:Npn \toks_use_gclear:N {
4666 <check> \pref_global_chk:
4667 <!check> \pref_global:D
4668 \toks_use_clear:N
4669 }

```

```

4670 \cs_generate_variant:Nn \toks_use_clear:N {c}
4671 \cs_generate_variant:Nn \toks_use_gclear:N {c}

```

(End definition for `\toks_use_clear:N`. This function is documented on page 88.)

`\toks_show:N` This function shows the contents of a token register on the terminal.
`\toks_show:c`

```

4672 \cs_new_eq:NN \toks_show:N \kernel_register_show:N
4673 \cs_generate_variant:Nn \toks_show:N {c}

```

(End definition for `\toks_show:N`. This function is documented on page 88.)

103.2 Adding to token registers' contents

`\toks_put_left:Nn` `\toks_put_left:Nn <toks><stuff>` adds the tokens of *stuff* on the ‘left-side’ of the token register *<toks>*. `\toks_put_left:No` does the same, but expands the tokens once. We need to look out for brace stripping so we add a token, which is then later removed.

```

4674 \cs_new_protected_nopar:Npn \toks_put_left:Nn #1 {
4675   \exp_after:wN \toks_put_left_aux:w \exp_after:wN \q_nil
4676   \toks_use:N #1 \q_stop #1
4677 }
\toks_gput_left:Nn
4678 \cs_generate_variant:Nn \toks_put_left:Nn {NV,No,Nx,cn,co,cV}
\toks_gput_left:NV
4679 \cs_new_protected_nopar:Npn \toks_gput_left:Nn {
4680   <check> \pref_global_chk:
\toks_gput_left:No
4681   <!check> \pref_global:D
\toks_gput_left:Nx
4682   \toks_put_left:Nn
\toks_gput_left:cn
4683 }
\toks_gput_left:cV
4684 \cs_generate_variant:Nn \toks_gput_left:Nn {NV,No,Nx,cn,cV,co}
\toks_gput_left:co
\toks_put_left_aux:w

```

A helper function for `\toks_put_left:Nn`. Its arguments are subsequently the tokens of *<stuff>*, the token register *<toks>* and the current contents of *<toks>*. We make sure to remove the token we inserted earlier.

```

4685 \cs_new:Npn \toks_put_left_aux:w #1\q_stop #2#3 {
4686   #2 \exp_after:wN { \use_i:nn {#3} #1 }
4687   <check> \chk_local_or_pref_global:N #2
4688 }

```

(End definition for `\toks_put_left:Nn`. This function is documented on page 89.)

`\toks_put_right:Nn` These macros add a list of tokens to the right of a token register.
`\toks_put_right:NV`
`\toks_put_right:No`
`\toks_put_right:Nx`
`\toks_put_right:cn`
`\toks_put_right:cV`
`\toks_put_right:co`
`\toks_gput_right:Nn`
`\toks_gput_right:NV`
`\toks_gput_right:No`
`\toks_gput_right:Nx`
`\toks_gput_right:cn`
`\toks_gput_right:cV`
`\toks_gput_right:co`


```

4693 \cs_new_protected_nopar:Npn \toks_gput_right:Nn {
4694   <check> \pref_global_chk:
4695   !<check> \pref_global:D
4696   \toks_put_right:Nn
4697 }

```

A couple done by hand for speed.

```

4698 (check) \cs_generate_variant:Nn \toks_put_right:Nn {No}
4699 {*\check}
4700 \cs_new_protected:Npn \toks_put_right:NV #1#2 {
4701   #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4702     \exp_after:wN \toks_use:N \exp_after:wN #1
4703     \int_to_roman:w -'0 \exp_eval_register:N #2
4704   }
4705 }
4706 \cs_new_protected:Npn \toks_put_right:No #1#2 {
4707   #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4708     \exp_after:wN \toks_use:N \exp_after:wN #1 #2
4709   }
4710 }
4711 {/\!check}
4712 \cs_generate_variant:Nn \toks_put_right:Nn {Nx,cn,cV,co}
4713 \cs_generate_variant:Nn \toks_gput_right:Nn {NV,No,Nx,cn,cV,co}

```

(End definition for \toks put right:Nn. This function is documented on page 89.)

`\toks_put_right:Nf` We implement `\toks_put_right:Nf` by hand because I think I might use it in the `l3keyval` module in which case it is going to be used a lot.

```

4714 {check}\cs_generate_variant:Nn \toks_put_right:Nn {Nf}
4715 {*!check}
4716 \cs_new_protected:Npn \toks_put_right:Nf #1#2 {
4717   #1 \exp_after:wN \exp_after:wN \exp_after:wN {
4718     \exp_after:wN \toks_use:N \exp_after:wN #1 \int_to_roman:w -'0#2
4719   }
4720 }
4721 </!check>

```

(End definition for `\toks_put_right:Nf`. This function is documented on page 89.)

103.3 Predicates and conditionals

<code>\toks_if_empty_p:N</code>	<code>\toks_if_empty:NTF</code>	$\langle \textit{toks} \rangle$	$\langle \textit{true code} \rangle$	$\langle \textit{false code} \rangle$	tests if a token register is empty and
<code>\toks_if_empty_p:c</code>					executes either $\langle \textit{true code} \rangle$ or $\langle \textit{false code} \rangle$. This test had the advantage of being expand-
<code>\toks_if_empty:NTF</code>					able. Otherwise one has to do an <code>x</code> type expansion in order to prevent problems with
<code>\toks_if_empty:cTF</code>					parameter tokens.

```

4722 \prg_new_conditional:Nnn \toks_if_empty:N {p,TF,T,F} {
4723   \tl_if_empty:VTF #1 {\prg_return_true:} {\prg_return_false:}
4724 }

```

```

4725 \cs_generate_variant:Nn \toks_if_empty_p:N {c}
4726 \cs_generate_variant:Nn \toks_if_empty:NTF {c}
4727 \cs_generate_variant:Nn \toks_if_empty:NT {c}
4728 \cs_generate_variant:Nn \toks_if_empty:NF {c}

```

(End definition for `\toks_if_empty_p:N` and `\toks_if_empty_p:c`. These functions are documented on page 90.)

```

\toks_if_eq_p:NN This function test whether two token registers have the same contents.
\toks_if_eq_p:cN
\toks_if_eq_p:Nc
\toks_if_eq_p:cc
\toks_if_eq:NNTF
\toks_if_eq:NcTF
\toks_if_eq:cNTF
\toks_if_eq:ccTF
4729 \prg_new_conditional:Nnn \toks_if_eq:NN {p,TF,T,F} {
4730   \str_if_eq:xxTF {\toks_use:N #1} {\toks_use:N #2}
4731   {\prg_return_true:} {\prg_return_false:}
4732 }
4733 \cs_generate_variant:Nn \toks_if_eq_p:NN {Nc,c,cc}
4734 \cs_generate_variant:Nn \toks_if_eq:NNTF {Nc,c,cc}
4735 \cs_generate_variant:Nn \toks_if_eq:NNT {Nc,c,cc}
4736 \cs_generate_variant:Nn \toks_if_eq:NNF {Nc,c,cc}

```

(End definition for `\toks_if_eq_p:NN` and others. These functions are documented on page 90.)

103.4 Variables and constants

`\l_tmpa_toks` Some scratch registers ...

```

\l_tmpb_toks
\l_tmpc_toks
\g_tmpa_toks
\g_tmpb_toks
\g_tmpc_toks
4737 \tex_toksdef:D \l_tmpa_toks = 255\scan_stop:
4738 \initex\seq_put_right:Nn \g_toks_allocation_seq {255}
4739 \toks_new:N \l_tmpb_toks
4740 \toks_new:N \l_tmpc_toks
4741 \toks_new:N \g_tmpa_toks
4742 \toks_new:N \g_tmpb_toks
4743 \toks_new:N \g_tmpc_toks

```

(End definition for `\l_tmpa_toks`. This function is documented on page 90.)

`\c_empty_toks` And here is a constant, which is a (permanently) empty token register.

```

4744 \toks_new:N \c_empty_toks

```

(End definition for `\c_empty_toks`. This function is documented on page 90.)

`\l_tl_replace_toks` And here is one for tl vars. Can't define it there as the allocation isn't set up at that point.

```

4745 \toks_new:N \l_tl_replace_toks

```

(End definition for `\l_tl_replace_toks`. This function is documented on page 90.)

```

4746 \</initex | package>

```

104 Internal sequence functions

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

Tests if the $\langle sequence \rangle$ is empty, and if so issues an error message before skipping over any tokens up to `\seq_break_point:n`. This function is used to avoid more serious errors which would otherwise occur if some internal functions were applied to an empty $\langle sequence \rangle$.

`\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:x` `\seq_push_item_def:n` $\{\langle code \rangle\}$

Saves the definition of `\seq_item:n` and redefines it to accept one parameter and expand to $\langle code \rangle$. This function should always be balanced by use of `\seq_pop_item_def:.`

`\seq_pop_item_def:` `\seq_pop_item_def:`

Restores the definition of `\seq_item:n` most recently saved by `\seq_push_item_def:n`. This function should always be used in a balanced pair with `\seq_push_item_def:n`.

`\seq_break: *` `\seq_break:`

Used to terminate sequence functions by gobbling all tokens up to `\seq_break_point:n`. This function is a copy of `\seq_map_break:`, but is used in situations which are not mappings.

`\seq_break:n *` `\seq_break:n` $\{\langle tokens \rangle\}$

Used to terminate sequence functions by gobbling all tokens up to `\seq_break_point:n`, then inserting the $\langle tokens \rangle$ before continuing reading the input stream. This function is a copy of `\seq_map_break:n`, but is used in situations which are not mappings.

`\seq_break_point:n *` `\seq_break_point:n` $\langle tokens \rangle$

Used to mark the end of a recursion or mapping: the functions `\seq_map_break:` and `\seq_map_break:n` use this to break out of the loop. After the loop ends the $\langle tokens \rangle$ are inserted into the input stream. This occurs even if the the break functions are *not* applied: `\seq_break_point:n` is functionally-equivalent in these cases to `\use:n`.

105 Sequence implementation

The following test files are used for this code: *m3seq002,m3seq003*.

```
4747 <*initex | package>
4748 <*package>
4749 \ProvidesExplPackage
4750   {\filename}{\filedate}{\fileversion}{\filedescription}
4751   \package_check_loaded_expl:
4752 </package>
```

A sequence is a control sequence whose top-level expansion is of the form “`\seq_item:n {<item1>} ... \seq_item:n {<itemn>}`”. An earlier implementation used the structure “`\seq_elt:w <item1> \seq_elt_end: ... \seq_elt:w <itemn> \seq_elt_end:`”. This allows rapid searching using a delimited function, but is not suitable for items containing `{`, `}` and `#` tokens, and which also leads to the loss of surrounding braces around items.

`\seq_item:n` The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

```
4753 \cs_new:Npn \seq_item:n
4754 {
4755   \seq_use_error:
4756   \use_none:n
4757 }
```

(End definition for `\seq_item:n`. This function is documented on page 324.)

`\l_seq_tmpa_tl` Scratch space for various internal uses.

```
\l_seq_tmpb_tl
4758 \tl_new:N \l_seq_tmpa_tl
4759 \tl_new:N \l_seq_tmpb_tl
```

(End definition for `\l_seq_tmpa_tl` and `\l_seq_tmpb_tl`.)

105.1 Allocation and initialisation

`\seq_new:N` Internally, sequences are just token lists.

`\seq_new:c`

```
4760 \cs_new_eq:NN \seq_new:N \tl_new:N
4761 \cs_new_eq:NN \seq_new:c \tl_new:c
```

(End definition for `\seq_new:N` and `\seq_new:c`. These functions are documented on page 91.)

`\seq_clear:N` Clearing sequences is just the same as clearing token lists.

```
\seq_clear:c
\seq_gclear:N
\seq_gclear:c
4762 \cs_new_eq:NN \seq_clear:N \tl_clear:N
4763 \cs_new_eq:NN \seq_clear:c \tl_clear:c
4764 \cs_new_eq:NN \seq_gclear:N \tl_gclear:N
4765 \cs_new_eq:NN \seq_gclear:c \tl_gclear:c
```

(End definition for `\seq_clear:N` and `\seq_clear:c`. These functions are documented on page 91.)

`\seq_clear_new:N` Once again a simply copy from the token list functions.
`\seq_clear_new:c`
`\seq_gclear_new:N`
`\seq_gclear_new:c`

```

4766 \cs_new_eq:NN \seq_clear_new:N \tl_clear_new:N
4767 \cs_new_eq:NN \seq_clear_new:c \tl_clear_new:c
4768 \cs_new_eq:NN \seq_gclear_new:N \tl_gclear_new:N
4769 \cs_new_eq:NN \seq_gclear_new:c \tl_gclear_new:c

```

(End definition for `\seq_clear_new:N` and `\seq_clear_new:c`. These functions are documented on page 91.)

`\seq_set_eq:NN` Once again, these are simply copies from the token list functions.
`\seq_set_eq:cN`
`\seq_set_eq:Nc`
`\seq_set_eq:cc`
`\seq_gset_eq:NN`
`\seq_gset_eq:cN`
`\seq_gset_eq:Nc`
`\seq_gset_eq:cc`

```

4770 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
4771 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
4772 \cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN
4773 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
4774 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
4775 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
4776 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
4777 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc

```

(End definition for `\seq_set_eq:NN` and others. These functions are documented on page 92.)

`\seq_concat:NNN` Concatenating sequences is easy.
`\seq_concat:ccc`
`\seq_gconcat:NNN`
`\seq_gconcat:ccc`

```

4778 \cs_new_protected_nopar:Npn \seq_concat:NNN #1#2#3
4779 { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4780 \cs_new_protected_nopar:Npn \seq_gconcat:NNN #1#2#3
4781 { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4782 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
4783 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_concat:ccc`. These functions are documented on page 92.)

105.2 Appending data to either end

`\seq_put_left:Nn` The code here is just a wrapper for adding to token lists.
`\seq_put_left:NV`
`\seq_put_left:Nv`
`\seq_put_left:No`
`\seq_put_left:Nx`
`\seq_put_left:cn`
`\seq_put_left:cV`
`\seq_put_left:cv`
`\seq_put_left:co`
`\seq_put_left:cx`
`\seq_put_right:Nn`
`\seq_put_right:NV`
`\seq_put_right:Nv`
`\seq_put_right:No`
`\seq_put_right:Nx`
`\seq_put_right:cn`
`\seq_put_right:cV`
`\seq_put_right:cv`
`\seq_put_right:co`

```

4784 \cs_new_protected:Npn \seq_put_left:Nn #1#2
4785 { \tl_put_left:Nn #1 { \seq_item:n {#2} } }
4786 \cs_new_protected:Npn \seq_put_right:Nn #1#2
4787 { \tl_put_right:Nn #1 { \seq_item:n {#2} } }
4788 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
4789 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
4790 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
4791 \cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , co , cx }

```

(End definition for `\seq_put_left:Nn` and others. These functions are documented on page 93.)

`\seq_gput_left:Nn` The same for global addition.

```

4792 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
4793 { \tl_gput_left:Nn #1 { \seq_item:n {#2} } }
4794 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
4795 { \tl_gput_right:Nn #1 { \seq_item:n {#2} } }
4796 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
4797 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }
4798 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
4799 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }

```

(End definition for `\seq_gput_left:Nn` and others. These functions are documented on page 93.)

105.3 Breaking sequence functions

To break a function, the special token `\seq_break_point:n` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

```

4800 \cs_new:Npn \seq_break: #1 \seq_break_point:n #2 {#2}
4801 \cs_new:Npn \seq_break:n #1#2 \seq_break_point:n #3 { #3 #1 }

```

(End definition for `\seq_break:.` This function is documented on page 324.)

`\seq_map_break:` Semantically-logical copies of the break functions for use inside mappings.

```

4802 \cs_new_eq:NN \seq_map_break: \seq_break:
4803 \cs_new_eq:NN \seq_map_break:n \seq_break:n

```

(End definition for `\seq_map_break:.` This function is documented on page 97.)

`\seq_break_point:n` Normally, the marker token will not be executed, but if it is then the end code is simply inserted.

```

4804 \cs_new_eq:NN \seq_break_point:n \use:n

```

(End definition for `\seq_break_point:n`. This function is documented on page 324.)

`\seq_if_empty_err_break:N` A function to check that sequences really have some content. This is optimised for speed, hence the direct primitive use.

```

4805 \cs_new_protected_nopar:Npn \seq_if_empty_err_break:N #1
4806 {
4807   \tex_ifx:D #1 \c_empty_tl
4808   \msg_kernel_bug:x { Empty-sequence-\token_to_str:N #1 }
4809   \exp_after:wN \seq_break:
4810   \tex_fi:D
4811 }

```

(End definition for `\seq_if_empty_err_break:N`. This function is documented on page 324.)

105.4 Mapping to sequences

`\seq_map_function:NN`
`\seq_map_function:cN`
`\seq_map_function_aux:NNn`

The idea here is to apply the code of #2 to each item in the sequence without altering the definition of `\seq_item:n`. This is done as by noting that every odd token in the sequence must be `\seq_item:n`, which can be gobbled by `\use_none:n`. At the end of the loop, #2 is instead ? `\seq_map_break:`, which therefore breaks the loop without needing to do a (relatively-expensive) quark test.

```

4812 \cs_new:Npn \seq_map_function:NN #1#2
4813 {
4814   \exp_after:wN \seq_map_function_aux:NNn \exp_after:wN #2 #1
4815   { ? \seq_map_break: } { }
4816   \seq_break_point:n { }
4817 }
4818 \cs_new:Npn \seq_map_function_aux:NNn #1#2#3
4819 {
4820   \use_none:n #2
4821   #1 {#3}
4822   \seq_map_function_aux:NNn #1
4823 }
4824 \cs_generate_variant:Nn \seq_map_function:NN { c }

```

(End definition for `\seq_map_function:NN` and `\seq_map_function:cN`. These functions are documented on page 96.)

`\g_seq_nesting_depth_int`

A counter to keep track of nested functions.

```

4825 \int_new:N \g_seq_nesting_depth_int

```

(End definition for `\g_seq_nesting_depth_int`.)

`\seq_push_item_def:n`
`\seq_push_item_def:x`
`\seq_push_item_def_aux:`
`\seq_pop_item_def:`

The definition of `\seq_item:n` needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```

4826 \cs_new_protected:Npn \seq_push_item_def:n
4827 {
4828   \seq_push_item_def_aux:
4829   \cs_gset:Npn \seq_item:n ##1
4830 }
4831 \cs_new_protected:Npn \seq_push_item_def:x
4832 {
4833   \seq_push_item_def_aux:
4834   \cs_gset:Npx \seq_item:n ##1
4835 }
4836 \cs_new_protected:Npn \seq_push_item_def_aux:
4837 {
4838   \cs_gset_eq:cN { seq_item_ \int_use:N \g_seq_nesting_depth_int :n }
4839   \seq_item:n
4840   \int_gincr:N \g_seq_nesting_depth_int

```

```

4841 }
4842 \cs_new_protected_nopar:Npn \seq_pop_item_def:
4843 {
4844   \int_gdecr:N \g_seq_nesting_depth_int
4845   \cs_gset_eq:Nc \seq_item:n
4846   { \seq_item_ \int_use:N \g_seq_nesting_depth_int :n }
4847 }

```

(End definition for `\seq_push_item_def:n` and `\seq_push_item_def:x`. These functions are documented on page 324.)

`\seq_map_inline:Nn` The idea here is that `\seq_item:n` is already “applied” to each item in a sequence, and
`\seq_map_inline:cn` so an in-line mapping is just a case of redefining `\seq_item:n`.

```

4848 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
4849 {
4850   \seq_push_item_def:n {#2}
4851   #1
4852   \seq_break_point:n { \seq_pop_item_def: }
4853 }
4854 \cs_generate_variant:Nn \seq_map_inline:Nn { c }

```

(End definition for `\seq_map_inline:Nn` and `\seq_map_inline:cn`. These functions are documented on page 96.)

`\seq_map_variable:NNn` This is just a specialised version of the in-line mapping function, using an x-type expansion
`\seq_map_variable:Ncn` for the code set up so that the number of # tokens required is as expected.
`\seq_map_variable:cNn`
`\seq_map_variable:ccn`

```

4855 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
4856 {
4857   \seq_push_item_def:x
4858   {
4859     \tl_set:Nn \exp_not:N #2 {##1}
4860     \exp_not:n {#3}
4861   }
4862   #1
4863   \seq_break_point:n { \seq_pop_item_def: }
4864 }
4865 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
4866 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }

```

(End definition for `\seq_map_variable:NNn` and others. These functions are documented on page 97.)

105.5 Sequence stacks

The same functions as for sequences, but with the correct naming.

`\seq_push:Nn` Pushing to a sequence is the same as adding on the left.
`\seq_push:NV`
`\seq_push:Nv`
`\seq_push:No`
`\seq_push:Nx`
`\seq_push:cn`
`\seq_push:cV`
`\seq_push:cV`
`\seq_push:co`
`\seq_push:cx`
`\seq_gpush:Nn`
`\seq_gpush:NV`


```

4867 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
4868 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
4869 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
4870 \cs_new_eq:NN \seq_push:No \seq_put_left:No
4871 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
4872 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
4873 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
4874 \cs_new_eq:NN \seq_push:cv \seq_put_left:cv
4875 \cs_new_eq:NN \seq_push:co \seq_put_left:co
4876 \cs_new_eq:NN \seq_push:cx \seq_put_left:cx
4877 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
4878 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
4879 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
4880 \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
4881 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
4882 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
4883 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
4884 \cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
4885 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
4886 \cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx

```

(End definition for `\seq_push:Nn` and others. These functions are documented on page 99.)

`\seq_get_left:NN` Getting an item from the left of a sequence is pretty easy: just trim off the first item
`\seq_get_left:cN` after removing the `\seq_item:n` at the start.
`\seq_get_left_aux:Nw`

```

4887 \cs_new_protected_nopar:Npn \seq_get_left:NN #1#2
4888 {
4889   \seq_if_empty_err_break:N #1
4890   \exp_after:wN \seq_get_left_aux:Nw #1 \q_stop #2
4891   \seq_break_point:n { }
4892 }
4893 \cs_new_protected:Npn \seq_get_left_aux:Nw \seq_item:n #1#2 \q_stop #3
4894 { \tl_set:Nn #3 {#1} }
4895 \cs_generate_variant:Nn \seq_get_left:NN { c }

```

(End definition for `\seq_get_left:NN` and `\seq_get_left:cN`. These functions are documented on page 94.)

`\seq_pop_left:NN` The approach to popping an item is pretty similar to that to get an item, with the only
`\seq_pop_left:cN` difference being that the sequence itself has to be redefined. This makes it more sensible
`\seq_gpop_left:NN` to use an auxiliary function for the local and global cases.
`\seq_gpop_left:cN`
`\seq_pop_left_aux:NNN`
`\seq_pop_left_aux:Nw`

```

4896 \cs_new_protected_nopar:Npn \seq_pop_left:NN
4897 { \seq_pop_left_aux:NNN \tl_set:Nn }
4898 \cs_new_protected_nopar:Npn \seq_gpop_left:NN
4899 { \seq_pop_left_aux:NNN \tl_gset:Nn }
4900 \cs_new_protected_nopar:Npn \seq_pop_left_aux:NNN #1#2#3
4901 {
4902   \seq_if_empty_err_break:N #2

```

```

4903     \exp_after:wN \seq_pop_left_aux:Nw #2 \q_stop #1#2#3
4904     \seq_break_point:n { }
4905   }
4906   \cs_new_protected:Npn \seq_pop_left_aux:Nw \seq_item:n #1#2 \q_stop #3#4#5
4907   {
4908     #3 #4 {#2}
4909     \tl_set:Nn #5 {#1}
4910   }
4911   \cs_generate_variant:Nn \seq_pop_left:NN { c }
4912   \cs_generate_variant:Nn \seq_gpop_left:NN { c }

```

(End definition for `\seq_pop_left:NN` and `\seq_pop_left:cN`. These functions are documented on page 94.)

`\seq_get_right:NN` The idea here is to remove the very first `\seq_item:n` from the sequence, leaving a token list starting with the first braced entry. Two arguments at a time are then grabbed: apart from the right-hand end of the sequence, this will be a brace group followed by `\seq_item:n`. The set up code means that these all disappear. At the end of the sequence, the assignment is placed in front of the very last entry in the sequence, before a tidying-up step takes place to remove the loop and reset the meaning of `\seq_item:n`.

`\seq_get_right:cN`
`\seq_get_right_aux:NN`
`\seq_get_right_loop:nn`

```

4913   \cs_new_protected_nopar:Npn \seq_get_right:NN #1#2
4914   {
4915     \seq_if_empty_err_break:N #1
4916     \seq_get_right_aux:NN #1#2
4917     \seq_break_point:n { }
4918   }
4919   \cs_new_protected_nopar:Npn \seq_get_right_aux:NN #1#2
4920   {
4921     \seq_push_item_def:n { }
4922     \exp_after:wN \exp_after:wN \exp_after:wN \seq_get_right_loop:nn
4923     \exp_after:wN \use_none:n #1
4924     { \tl_set:Nn #2 }
4925     { }
4926     {
4927       \seq_pop_item_def:
4928       \seq_break:
4929     }
4930   }
4931   \cs_new:Npn \seq_get_right_loop:nn #1#2
4932   {
4933     #2 {#1}
4934     \seq_get_right_loop:nn
4935   }
4936   \cs_generate_variant:Nn \seq_get_right:NN { c }

```

(End definition for `\seq_get_right:NN` and `\seq_get_right:cN`. These functions are documented on page 94.)

`\seq_pop_right:NN` The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a “flexible length” way
`\seq_pop_right:cN`
`\seq_gpop_right:NN`
`\seq_gpop_right:cN`

`\seq_get_right_aux:NNN`
`\seq_get_right_aux_ii:NNN`

to set a token list variable. This is supplied by the `{ \tex_iffalse:D } \tex_fi:D` ...`\tex_iffalse:D { \tex_fi:D }` construct. Using an x-type expansion and a “non-expanding” definition for `\seq_item:n`, the left-most $n-1$ entries in a sequence of n items will be stored back in the sequence. That needs a loop of unknown length, hence using the strange `\tex_iffalse:D` way of including brackets. When the last item of the sequence is reached, the closing bracket for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment, then a final loop clears up the code.

```

4937 \cs_new_protected_nopar:Npn \seq_pop_right:NN
4938 { \seq_pop_right_aux:NNN \tl_set:Nx }
4939 \cs_new_protected_nopar:Npn \seq_gpop_right:NN
4940 { \seq_pop_right_aux:NNN \tl_gset:Nx }
4941 \cs_new_protected_nopar:Npn \seq_pop_right_aux:NNN #1#2#3
4942 {
4943   \seq_if_empty_err_break:N #2
4944   \seq_pop_right_aux_ii:NNN #1 #2 #3
4945   \seq_break_point:n { }
4946 }
4947 \cs_new_protected_nopar:Npn \seq_pop_right_aux_ii:NNN #1#2#3
4948 {
4949   \seq_push_item_def:n { \exp_not:n { \seq_item:n {##1} } }
4950   #1 #2 { \tex_iffalse:D } \tex_fi:D
4951   \exp_after:wN \exp_after:wN \exp_after:wN \seq_get_right_loop:nn
4952   \exp_after:wN \use_none:n #2
4953   {
4954     \tex_iffalse:D { \tex_fi:D }
4955     \tl_set:Nn #3
4956   }
4957   { }
4958   {
4959     \seq_pop_item_def:
4960     \seq_break:
4961   }
4962 }
4963 \cs_generate_variant:Nn \seq_pop_right:NN { c }
4964 \cs_generate_variant:Nn \seq_gpop_right:NN { c }

```

(End definition for `\seq_pop_right:NN` and `\seq_pop_right:cN`. These functions are documented on page 94.)

`\seq_get:NN` In most cases, getting items from the stack does not need to specify that this is from the
`\seq_get:cN` left. So alias are provided.

```

\seq_pop:NN
\seq_pop:cN
\seq_gpop:NN
\seq_gpop:cN
4965 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
4966 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
4967 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
4968 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
4969 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
4970 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN

```

(End definition for `\seq_get:Nn` and `\seq_get:cN`. These functions are documented on page 98.)

105.6 Sequence conditionals

`\seq_if_empty_p:N` Simple copies from the token list variable material.

```

\seq_if_empty_p:c 4971 \prg_new_eq_conditional:Nn \seq_if_empty:N \tl_if_empty:N
\seq_if_empty:N $\underline{TF}$  4972 { p , T , F , TF }
\seq_if_empty:c $\underline{TF}$  4973 \prg_new_eq_conditional:Nn \seq_if_empty:c \tl_if_empty:c
4974 { p , T , F , TF }
```

(End definition for `\seq_if_empty:N` and `\seq_if_empty:c`. These functions are documented on page 96.)

`\seq_if_in:Nn \underline{TF}` The approach here is to define `\seq_item:n` to compare its argument with the test
`\seq_if_in:Nv \underline{TF}` sequence. If the two items are equal, the mapping is terminated and `\prg_return_`
`\seq_if_in:Nv \underline{TF}` `true:` is inserted. On the other hand, if there is no match then the loop will break
`\seq_if_in:No \underline{TF}` returning `\prg_return_false:`. In either case, `\seq_break_point:n` ensures that the
`\seq_if_in:Nx \underline{TF}` group ends before the logical value is returned. Everything is inside a group so that
`\seq_if_in:cn \underline{TF}` `\seq_item:n` is preserved in nested situations.
`\seq_if_in:cV \underline{TF}`

```

\seq_if_in:cV $\underline{TF}$  4975 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
\seq_if_in:cv $\underline{TF}$  4976 { T , F , TF }
\seq_if_in:co $\underline{TF}$  4977 {
\seq_if_in:cx $\underline{TF}$  4978   \group_begin:
\seq_if_in_aux: 4979   \tl_set:Nn \l_seq_tmpa_tl {#2}
4980   \cs_set_protected:Npn \seq_item:n ##1
4981   {
4982     \tl_set:Nn \l_seq_tmpb_tl {##1}
4983     \tex_ifx:D \l_seq_tmpa_tl \l_seq_tmpb_tl
4984     \exp_after:wN \seq_if_in_aux:
4985     \tex_fi:D
4986   }
4987   #1
4988   \seq_break:n { \prg_return_false: }
4989   \seq_break_point:n { \group_end: }
4990 }
4991 \cs_new_nopar:Npn \seq_if_in_aux: { \seq_break:n { \prg_return_true: } }
4992 \cs_generate_variant:Nn \seq_if_in:NnT { NV , Nv , No , Nx }
4993 \cs_generate_variant:Nn \seq_if_in:NnT { c , cV , cv , co , cx }
4994 \cs_generate_variant:Nn \seq_if_in:NnF { NV , Nv , No , Nx }
4995 \cs_generate_variant:Nn \seq_if_in:NnF { c , cV , cv , co , cx }
4996 \cs_generate_variant:Nn \seq_if_in:NnTF { NV , Nv , No , Nx }
4997 \cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }
```

(End definition for `\seq_if_in:Nn` and others. These functions are documented on page 96.)

105.7 Modifying sequences

`\l_seq_remove_seq` An internal sequence for the removal routines.

```
4998 \seq_new:N \l_seq_remove_seq
```

(End definition for `\l_seq_remove_seq`.)

`\seq_remove_duplicates:N` Removing duplicates means making a new list then copying it.

`\seq_remove_duplicates:c`

`\seq_gremove_duplicates:N`

`\seq_gremove_duplicates:c`

`\seq_remove_duplicates_aux:NN`

```
4999 \cs_new_protected:Npn \seq_remove_duplicates:N
5000 { \seq_remove_duplicates_aux:NN \seq_set_eq:NN }
5001 \cs_new_protected:Npn \seq_gremove_duplicates:N
5002 { \seq_remove_duplicates_aux:NN \seq_gset_eq:NN }
5003 \cs_new_protected:Npn \seq_remove_duplicates_aux:NN #1#2
5004 {
5005   \seq_clear:N \l_seq_remove_seq
5006   \seq_map_inline:Nn #2
5007   {
5008     \seq_if_in:NnF \l_seq_remove_seq {##1}
5009     { \seq_put_right:Nn \l_seq_remove_seq {##1} }
5010   }
5011   #1 #2 \l_seq_remove_seq
5012 }
5013 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
5014 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }
```

(End definition for `\seq_remove_duplicates:N` and `\seq_remove_duplicates:c`. These functions are documented on page 95.)

`\seq_remove_all:Nn`

`\seq_remove_all:cn`

`\seq_gremove_all:Nn`

`\seq_gremove_all:cn`

`\seq_remove_all_aux:NNn`

The idea of the code here is to avoid a relatively expensive addition of items one at a time to an intermediate sequence. The approach taken is therefore similar to that in `\seq_pop_right_aux_ii:NNN`, using a “flexible” x-type expansion to do most of the work. As `\tl_if_eq:nnT` is not expandable, a two-part strategy is needed. First, the x-type expansion uses `\str_if_eq:nnT` to find potential matches. If one is found, the expansion is halted and the necessary set up takes place to use the `\tl_if_eq:nnT` test. The x-type is started again, including all of the items copied already. This will happen repeatedly until the entire sequence has been scanned. The code is set up to avoid needing and intermediate scratch list: the lead-off x-type expansion (`#1 #2 {#2}`) will ensure that nothing is lost.

```
5015 \cs_new_protected:Npn \seq_remove_all:Nn
5016 { \seq_remove_all_aux:NNn \tl_set:Nx }
5017 \cs_new_protected:Npn \seq_gremove_all:Nn
5018 { \seq_remove_all_aux:NNn \tl_gset:Nx }
5019 \cs_new_protected:Npn \seq_remove_all_aux:NNn #1#2#3
5020 {
5021   \seq_push_item_def:n
5022   {
5023     \str_if_eq:nnT {##1} {##3}
```

```

5024         {
5025             \tex_iffalse:D { \tex_fi:D }
5026             \tl_set:Nn \l_seq_tmpb_tl {##1}
5027             #1 #2
5028             { \tex_iffalse:D } \tex_fi:D
5029             \exp_not:o {#2}
5030             \tl_if_eq:NNT \l_seq_tmpa_tl \l_seq_tmpb_tl
5031             { \use_none:nn }
5032         }
5033         \exp_not:n { \seq_item:n {##1} }
5034     }
5035     \tl_set:Nn \l_seq_tmpa_tl {#3}
5036     #1 #2 {#2}
5037     \seq_pop_item_def:
5038 }
5039 \cs_generate_variant:Nn \seq_remove_all:Nn { c }
5040 \cs_generate_variant:Nn \seq_gremove_all:Nn { c }

```

(End definition for `\seq_remove_all:Nn` and `\seq_remove_all:cn`. These functions are documented on page 95.)

106 Viewing sequences

`\l_seq_show_tl` Used to store the material for display.

```

5041 \tl_new:N \l_seq_show_tl

```

(End definition for `\l_seq_show_tl`.)

`\seq_show:N` The aim of the mapping here is to create a token list containing the formatted sequence.
`\seq_show:c` The very first item needs the new line and `>_` removing, which is achieved using a `w`-type
`\seq_show_aux:n` auxiliary. To avoid a low-level TeX error if there is an empty sequence, a simple test is
`\seq_show_aux:w` used to keep the output “clean”.

```

5042 \cs_new_protected_nopar:Npn \seq_show:N #1
5043 {
5044     \seq_if_empty:NTF #1
5045     {
5046         \iow_term:x { Sequence~\token_to_str:N #1 \c_space_tl is-empty }
5047         \tl_show:n { }
5048     }
5049     {
5050         \iow_term:x
5051         {
5052             Sequence~\token_to_str:N #1 \c_space_tl
5053             contains~the~items~(without~outer~braces):
5054         }
5055         \tl_set:Nx \l_seq_show_tl

```

```

5056         { \seq_map_function:NN #1 \seq_show_aux:n }
5057         \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
5058         { \exp_after:wN \seq_show_aux:w \l_seq_show_tl }
5059     }
5060 }
5061 \cs_new:Npn \seq_show_aux:n #1
5062 {
5063     \iow_newline: > \c_space_tl \c_space_tl
5064     \iow_char:N \{ \exp_not:n {#1} \iow_char:N \}
5065 }
5066 \cs_new:Npn \seq_show_aux:w #1 > ~ { }
5067 \cs_generate_variant:Nn \seq_show:N { c }

```

(End definition for `\seq_show:N` and `\seq_show:c`. These functions are documented on page 99.)

106.1 Depreciated interfaces

A few functions which are no longer documented: these were moved here on or before 2011-04-20, and will be removed entirely by 2011-07-20.

```

\seq_top:NN These are old stack functions.
\seq_top:cN
5068 \cs_new_eq:NN \seq_top:NN \seq_get_left:NN
5069 \cs_new_eq:NN \seq_top:cN \seq_get_left:cN

```

(End definition for `\seq_top:NN` and `\seq_top:cN`. These functions are documented on page ??.)

```

\seq_display:N An older name for \seq_show:N.
\seq_display:c
5070 \cs_new_eq:NN \seq_display:N \seq_show:N
5071 \cs_new_eq:NN \seq_display:c \seq_show:c

```

(End definition for `\seq_display:N` and `\seq_display:c`. These functions are documented on page ??.)

```

5072 </initex | package>

```

107 l3clist implementation

The following test files are used for this code: `m3clist002.lvt`.

We start by ensuring that the required packages are loaded.

```

5073 <*package>
5074 \ProvidesExplPackage
5075 { \filename } { \filedate } { \fileversion } { \filedescription }
5076 \package_check_loaded_expl:
5077 </package>
5078 <*initex | package>

```

107.1 Allocation and initialisation

`\clist_new:N` Comma-Lists are implemented using token lists.

`\clist_new:c`

```
5079 \cs_new_eq:NN \clist_new:N \tl_new:N
5080 \cs_generate_variant:Nn \clist_new:N {c}
```

(End definition for `\clist_new:N` and `\clist_new:c`. These functions are documented on page 100.)

`\clist_clear:N` Clearing a comma-list is the same as clearing a token list.

`\clist_clear:c`

`\clist_gclear:N`

`\clist_gclear:c`

```
5081 \cs_new_eq:NN \clist_clear:N \tl_clear:N
5082 \cs_generate_variant:Nn \clist_clear:N {c}
5083 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
5084 \cs_generate_variant:Nn \clist_gclear:N {c}
```

(End definition for `\clist_clear:N` and `\clist_clear:c`. These functions are documented on page 101.)

`\clist_clear_new:N` Clearing a comma-list is the same as clearing a token list.

`\clist_clear_new:c`

`\clist_gclear_new:N`

`\clist_gclear_new:c`

```
5085 \cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N
5086 \cs_generate_variant:Nn \clist_clear_new:N {c}
5087 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
5088 \cs_generate_variant:Nn \clist_gclear_new:N {c}
```

(End definition for `\clist_clear_new:N` and `\clist_clear_new:c`. These functions are documented on page 101.)

`\clist_set_eq:NN` We can set one $\langle\textit{clist}\rangle$ equal to another.

`\clist_set_eq:cN`

`\clist_set_eq:Nc`

`\clist_set_eq:cc`

```
5089 \cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN
5090 \cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN
5091 \cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc
5092 \cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc
```

(End definition for `\clist_set_eq:NN` and others. These functions are documented on page 100.)

`\clist_gset_eq:NN` An of course globally which seems to be needed far more often.

`\clist_gset_eq:cN`

`\clist_gset_eq:Nc`

`\clist_gset_eq:cc`

```
5093 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
5094 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
5095 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
5096 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(End definition for `\clist_gset_eq:NN` and others. These functions are documented on page 100.)

107.2 Predicates and conditionals

`\clist_if_empty_p:N`
`\clist_if_empty_p:c`
`\clist_if_empty:N \overline{TF}`
`\clist_if_empty:c \overline{TF}`

5097 `\prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N {p,TF,T,F}`
5098 `\prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c {p,TF,T,F}`

(End definition for `\clist_if_empty_p:N` and `\clist_if_empty_p:c`. These functions are documented on page 104.)

`\clist_if_empty_err:N` Signals an error if the comma-list is empty.

5099 `\cs_new_protected_nopar:Npn \clist_if_empty_err:N #1 {`
5100 `\if_meaning:w #1 \c_empty_tl`
5101 `\tl_clear:N \l_kernel_testa_tl % catch prefixes`
5102 `\msg_kernel_bug:x {Empty~comma-list~'\token_to_str:N #1'}`
5103 `\fi:`
5104 `}`

(End definition for `\clist_if_empty_err:N`. This function is documented on page 107.)

`\clist_if_eq_p:NN`
`\clist_if_eq_p:Nc`
`\clist_if_eq_p:cN`
`\clist_if_eq_p:cc`
`\clist_if_eq:NN \overline{TF}`
`\clist_if_eq:cN \overline{TF}`
`\clist_if_eq:Nc \overline{TF}`
`\clist_if_eq:cc \overline{TF}`

Returns `\c_true` iff the two comma-lists are equal.

5105 `\prg_new_eq_conditional:NNn \clist_if_eq:NN \tl_if_eq:NN {p,TF,T,F}`
5106 `\prg_new_eq_conditional:NNn \clist_if_eq:cN \tl_if_eq:cN {p,TF,T,F}`
5107 `\prg_new_eq_conditional:NNn \clist_if_eq:Nc \tl_if_eq:Nc {p,TF,T,F}`
5108 `\prg_new_eq_conditional:NNn \clist_if_eq:cc \tl_if_eq:cc {p,TF,T,F}`

(End definition for `\clist_if_eq_p:NN` and others. These functions are documented on page 105.)

`\clist_if_in:Nn \overline{TF}`
`\clist_if_in:NV \overline{TF}`
`\clist_if_in:No \overline{TF}`
`\clist_if_in:cn \overline{TF}`
`\clist_if_in:cV \overline{TF}`
`\clist_if_in:co \overline{TF}`

`\clist_if_in:Nn \overline{TF}` $\langle clist \rangle \langle item \rangle \langle true\ case \rangle \langle false\ case \rangle$ will check whether $\langle item \rangle$ is in $\langle clist \rangle$ and then either execute the $\langle true\ case \rangle$ or the $\langle false\ case \rangle$. $\langle true\ case \rangle$ and $\langle false\ case \rangle$ may contain incomplete `\if_charcode:w` statements.

5109 `\prg_new_protected_conditional:Nnn \clist_if_in:Nn {TF,T,F} {`
5110 `\cs_set:Npn \clist_tmp:w ##1,##2,##3 \q_stop {`
5111 `\if_meaning:w \q_no_value ##2`
5112 `\prg_return_false: \else: \prg_return_true: \fi:`
5113 `}`
5114 `\exp_last_unbraced:NNo \clist_tmp:w , #1 , #2 , \q_no_value \q_stop`
5115 `}`

5116 `\cs_generate_variant:Nn \clist_if_in:Nn \overline{TF} {NV,No,cn,cV,co}`
5117 `\cs_generate_variant:Nn \clist_if_in:NnT {NV,No,cn,cV,co}`
5118 `\cs_generate_variant:Nn \clist_if_in:NnF {NV,No,cn,cV,co}`

(End definition for `\clist_if_in:Nn`. This function is documented on page 105.)

107.3 Retrieving data

`\clist_use:N` Using a $\langle clist \rangle$ is just executing it but if $\langle clist \rangle$ equals `\scan_stop:` it is probably stemming from a `\cs:w ... \cs_end:` that was created by mistake somewhere.

`\clist_use:c`

```

5119 \cs_new_nopar:Npn \clist_use:N #1 {
5120   \if_meaning:w #1 \scan_stop:
5121     \msg_kernel_bug:x {
5122       Comma~list~ '\token_to_str:N #1'~ has~ an~ erroneous~ structure!}
5123   \else:
5124     \exp_after:wN #1
5125   \fi:
5126 }
5127 \cs_generate_variant:Nn \clist_use:N {c}

```

(End definition for `\clist_use:N` and `\clist_use:c`. These functions are documented on page 102.)

`\clist_get:NN` $\langle comma-list \rangle \langle cmd \rangle$ defines $\langle cmd \rangle$ to be the left-most element of $\langle comma-list \rangle$.

`\clist_get:cN`

```

5128 \cs_new_protected_nopar:Npn \clist_get:NN #1 {
5129   \clist_if_empty_err:N #1
5130   \exp_after:wN \clist_get_aux:w #1,\q_stop
5131 }
5132 \cs_generate_variant:Nn \clist_get:NN {cN}

```

(End definition for `\clist_get:NN` and `\clist_get:cN`. These functions are documented on page 103.)

`\clist_get_aux:w`

```

5133 \cs_new_protected:Npn \clist_get_aux:w #1,#2\q_stop #3 { \tl_set:Nn #3{#1} }

```

(End definition for `\clist_get_aux:w`.)

`\clist_pop_aux:nnNN` $\langle def_1 \rangle \langle def_2 \rangle \langle comma-list \rangle \langle cmd \rangle$ assigns the left-most element of $\langle comma-list \rangle$ to $\langle cmd \rangle$ using $\langle def_2 \rangle$, and assigns the tail of $\langle comma-list \rangle$ to $\langle comma-list \rangle$ using $\langle def_1 \rangle$.

`\clist_pop_aux:w`

`\clist_pop_auxi:w`

```

5134 \cs_new_protected:Npn \clist_pop_aux:nnNN #1#2#3 {
5135   \clist_if_empty_err:N #3
5136   \exp_after:wN \clist_pop_aux:w #3,\q_nil\q_stop #1#2#3
5137 }

```

After the assignments below, if there was only one element in the original `clist`, it now contains only `\q_nil`.

```

5138 \cs_new_protected:Npn \clist_pop_aux:w #1,#2\q_stop #3#4#5#6 {
5139   #4 #6 {#1}
5140   #3 #5 {#2}
5141   \quark_if_nil:NTF #5 { #3 #5 {} }{ \clist_pop_auxi:w #2 #3#5 }
5142 }

```

```
5143 \cs_new:Npn \clist_pop_auxi:w #1,\q_nil #2#3 { #2#3{#1} }
```

(End definition for `\clist_pop_aux:nnNN`.)

```
\clist_show:N
\clist_show:c
```

```
5144 \cs_new_eq:NN \clist_show:N \tl_show:N
```

```
5145 \cs_new_eq:NN \clist_show:c \tl_show:c
```

(End definition for `\clist_show:N`. This function is documented on page 102.)

```
\clist_display:N
\clist_display:c
```

```
5146 \cs_new_protected_nopar:Npn \clist_display:N #1 {
5147   \iow_term:x { Comma-list~\token_to_str:N #1~contains~
5148     the~elements~(without~outer~braces): }
5149   \toks_clear:N \l_tmpa_toks
5150   \clist_map_inline:Nn #1 {
5151     \toks_if_empty:NF \l_tmpa_toks {
5152       \toks_put_right:Nx \l_tmpa_toks {^^J>~}
5153     }
5154     \toks_put_right:Nx \l_tmpa_toks {
5155       \c_space_tl \iow_char:N \{ \exp_not:n {##1} \iow_char:N \}
5156     }
5157   }
5158   \toks_show:N \l_tmpa_toks
5159 }
5160 \cs_generate_variant:Nn \clist_display:N {c}
```

(End definition for `\clist_display:N`. This function is documented on page 103.)

107.4 Storing data

`\clist_put_aux:NNnnNn` The generic put function. When adding we have to distinguish between an empty `\clist` and one that contains at least one item (otherwise we accumulate commas).

MH says: Perhaps we should make sure that empty arguments don't get on the stack as that is probably a mistake. That's what I've implemented here. Since `\tl_if_empty:nF` is expandable prefixes are still allowed.

```
5161 \cs_new_protected:Npn \clist_put_aux:NNnnNn #1#2#3#4#5#6 {
5162   \clist_if_empty:NTF #5 { #1 #5 {#6} } {
5163     \tl_if_empty:nF {#6} { #2 #5{#3#6#4} }
5164   }
5165 }
```

(End definition for `\clist_put_aux:NNnnNn`.)

`\clist_put_left:Nn` The operations for adding to the left.
`\clist_put_left:NV`
`\clist_put_left:No`
`\clist_put_left:Nx`
`\clist_put_left:cn`
`\clist_put_left:cV`
`\clist_put_left:co`

```

5166 \cs_new_protected_nopar:Npn \clist_put_left:Nn {
5167   \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_left:Nn {} ,
5168 }
5169 \cs_generate_variant:Nn \clist_put_left:Nn {NV,No,Nx,cn,cV,co}

```

(End definition for `\clist_put_left:Nn` and others. These functions are documented on page 101.)

`\clist_gput_left:Nn` Global versions.
`\clist_gput_left:NV`
`\clist_gput_left:No`
`\clist_gput_left:Nx`
`\clist_gput_left:cn`
`\clist_gput_left:cV`
`\clist_gput_left:co`

```

5170 \cs_new_protected_nopar:Npn \clist_gput_left:Nn {
5171   \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_left:Nn {} ,
5172 }
5173 \cs_generate_variant:Nn \clist_gput_left:Nn {NV,No,Nx,cn,cV,co}

```

(End definition for `\clist_gput_left:Nn` and others. These functions are documented on page 102.)

`\clist_put_right:Nn` Adding something to the right side is almost the same.
`\clist_put_right:NV`
`\clist_put_right:No`
`\clist_put_right:Nx`
`\clist_put_right:cn`
`\clist_put_right:cV`
`\clist_put_right:co`

```

5174 \cs_new_protected_nopar:Npn \clist_put_right:Nn {
5175   \clist_put_aux:NNnnNn \tl_set:Nn \tl_put_right:Nn , {}
5176 }
5177 \cs_generate_variant:Nn \clist_put_right:Nn {NV,No,Nx,cn,cV,co}

```

(End definition for `\clist_put_right:Nn` and others. These functions are documented on page 102.)

`\clist_gput_right:Nn` And here the global variants.
`\clist_gput_right:NV`
`\clist_gput_right:No`
`\clist_gput_right:Nx`
`\clist_gput_right:cn`
`\clist_gput_right:cV`
`\clist_gput_right:co`

```

5178 \cs_new_protected_nopar:Npn \clist_gput_right:Nn {
5179   \clist_put_aux:NNnnNn \tl_gset:Nn \tl_gput_right:Nn , {}
5180 }
5181 \cs_generate_variant:Nn \clist_gput_right:Nn {NV,No,Nx,cn,cV,co}

```

(End definition for `\clist_gput_right:Nn` and others. These functions are documented on page 102.)

107.5 Mapping

`\clist_map_function:NN` Using the above creating the comma mappings is easy..
`\clist_map_function:Nc`
`\clist_map_function:cN`
`\clist_map_function:cc`
`\clist_map_function:nN`
`\clist_map_function:nc`
`\clist_map_inline:Nn`
`\clist_map_inline:nn`
`\clist_map_break:`

```

5182 \prg_new_map_functions:Nn , { clist }
5183 \cs_generate_variant:Nn \clist_map_function:NN { Nc }
5184 \cs_generate_variant:Nn \clist_map_function:NN { c }
5185 \cs_generate_variant:Nn \clist_map_function:NN { cc }
5186 \cs_generate_variant:Nn \clist_map_inline:Nn { c }
5187 \cs_generate_variant:Nn \clist_map_inline:Nn { nc }

```

(End definition for `\clist_map_function:NN` and others. These functions are documented on page 104.)

`\clist_map_variable:nNn` `\clist_map_variable:NNn` `\clist_map_variable:cNn` $\langle comma-list \rangle$ $\langle temp \rangle$ $\langle action \rangle$ assigns $\langle temp \rangle$ to each element and executes $\langle action \rangle$.

```

5188 \cs_new_protected:Npn \clist_map_variable:nNn #1#2#3 {
5189   \tl_if_empty:nF {#1} {
5190     \clist_map_variable_aux:Nnw #2 {#3} #1
5191     , \q_recursion_tail , \q_recursion_stop
5192   }
5193 }

```

Something for v/V

```

5194 \cs_new_protected_nopar:Npn \clist_map_variable:NNn {\exp_args:No \clist_map_variable:nNn}
5195 \cs_generate_variant:Nn\clist_map_variable:NNn {cNn}

```

(End definition for `\clist_map_variable:nNn`, `\clist_map_variable:NNn`, and `\clist_map_variable:cNn`. These functions are documented on page 104.)

`\clist_map_variable_aux:Nnw` The general loop. Assign the temp variable #1 to the current item #3 and then check if that's the stop marker. If it is, break the loop. If not, execute the action #2 and continue.

```

5196 \cs_new_protected:Npn \clist_map_variable_aux:Nnw #1#2#3,{
5197   \cs_set_nopar:Npn #1{#3}
5198   \quark_if_recursion_tail_stop:N #1
5199   #2 \clist_map_variable_aux:Nnw #1{#2}
5200 }

```

(End definition for `\clist_map_variable_aux:Nnw`.)

107.6 Higher level functions

`\clist_concat:NNN` `\clist_concat:ccc` `\clist_gconcat:NNN` `\clist_gconcat:ccc` `\clist_concat_aux:NNNN` $\langle clist 1 \rangle$ $\langle clist 2 \rangle$ $\langle clist 3 \rangle$ will globally assign $\langle clist 1 \rangle$ the concatenation of $\langle clist 2 \rangle$ and $\langle clist 3 \rangle$.
Again the situation is a bit more complicated because of the use of commas between items, so if either list is empty we have to avoid adding a comma.

```

5201 \cs_new_protected_nopar:Npn \clist_concat_aux:NNNN #1#2#3#4 {
5202   \tl_set:No \l_tmpa_tl {#3}
5203   \tl_set:No \l_tmpb_tl {#4}
5204   #1 #2 {
5205     \exp_not:V \l_tmpa_tl
5206     \tl_if_empty:NF \l_tmpa_tl { \tl_if_empty:NF \l_tmpb_tl , }
5207     \exp_not:V \l_tmpb_tl
5208   }
5209 }
5210 \cs_new_protected_nopar:Npn \clist_concat:NNN { \clist_concat_aux:NNNN \tl_set:Nx }
5211 \cs_new_protected_nopar:Npn \clist_gconcat:NNN { \clist_concat_aux:NNNN \tl_gset:Nx }
5212 \cs_generate_variant:Nn \clist_concat:NNN {ccc}
5213 \cs_generate_variant:Nn \clist_gconcat:NNN {ccc}

```

(End definition for `\clist_concat:NNN` and `\clist_concat:ccc`. These functions are documented on page 105.)

`\l_clist_remove_clist` A common scratch space for the removal routines.

5214 `\clist_new:N \l_clist_remove_clist`

(End definition for `\l_clist_remove_clist`.)

`\clist_remove_duplicates_aux:NN`

`\clist_remove_duplicates_aux:n`

Removing duplicate entries in a `\clist` is fairly straight forward. We use a temporary variable and then go through the list from left to right. For each element check if the element is already present in the list.

`\clist_remove_duplicates:N`

`\clist_gremove_duplicates:N`

```
5215 \cs_new_protected:Npn \clist_remove_duplicates_aux:NN #1#2 {
5216   \clist_clear:N \l_clist_remove_clist
5217   \clist_map_function:NN #2 \clist_remove_duplicates_aux:n
5218   #1 #2 \l_clist_remove_clist
5219 }
5220 \cs_new_protected:Npn \clist_remove_duplicates_aux:n #1 {
5221   \clist_if_in:NnF \l_clist_remove_clist {#1} {
5222     \clist_put_right:Nn \l_clist_remove_clist {#1}
5223   }
5224 }
```

The high level functions are just for telling if it should be a local or global setting.

```
5225 \cs_new_protected_nopar:Npn \clist_remove_duplicates:N {
5226   \clist_remove_duplicates_aux:NN \clist_set_eq:NN
5227 }
5228 \cs_new_protected_nopar:Npn \clist_gremove_duplicates:N {
5229   \clist_remove_duplicates_aux:NN \clist_gset_eq:NN
5230 }
```

(End definition for `\clist_remove_duplicates_aux:NN`.)

`\clist_remove_element:Nn`

`\clist_gremove_element:Nn`

`\clist_remove_element_aux:NNn`

The same general idea is used for removing elements: the parent functions just set things up for the internal ones.

`\clist_remove_element_aux:n`

```
5231 \cs_new_protected_nopar:Npn \clist_remove_element:Nn {
5232   \clist_remove_element_aux:NNn \clist_set_eq:NN
5233 }
5234 \cs_new_protected_nopar:Npn \clist_gremove_element:Nn {
5235   \clist_remove_element_aux:NNn \clist_gset_eq:NN
5236 }
5237 \cs_new_protected:Npn \clist_remove_element_aux:NNn #1#2#3 {
5238   \clist_clear:N \l_clist_remove_clist
5239   \cs_set:Npn \clist_remove_element_aux:n ##1 {
5240     \str_if_eq:nnF {#3} {##1} {
5241       \clist_put_right:Nn \l_clist_remove_clist {##1}
5242     }
5243   }
```

```

5243 }
5244 \clist_map_function:NN #2 \clist_remove_element_aux:n
5245 #1 #2 \l_clist_remove_clist
5246 }
5247 \cs_new:Npn \clist_remove_element_aux:n #1 { }

```

(End definition for `\clist_remove_element:Nm`. This function is documented on page 106.)

107.7 Stack operations

We build stacks from comma-lists, but here we put the specific functions together.

`\clist_push:Nn` Since comma-lists can be used as stacks, we ought to have both ‘push’ and ‘pop’. In most cases they are nothing more than new names for old functions.

`\clist_push:No`

`\clist_push:NV`

`\clist_push:cn`

`\clist_pop:NN`

`\clist_pop:cN`

```

5248 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn
5249 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV
5250 \cs_new_eq:NN \clist_push:No \clist_put_left:No
5251 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn
5252 \cs_new_protected_nopar:Npn \clist_pop:NN {\clist_pop_aux:nnNN \tl_set:Nn \tl_set:Nn}
5253 \cs_generate_variant:Nn \clist_pop:NN {cN}

```

(End definition for `\clist_push:Nn` and others. These functions are documented on page 107.)

`\clist_gpush:Nn` I don’t agree with Denys that one needs only local stacks, actually I believe that one will probably need the functions here more often. In case of `\clist_gpop:NN` the value is nevertheless returned locally.

`\clist_gpush:No`

`\clist_gpush:NV`

`\clist_gpush:cn`

`\clist_gpop:NN`

`\clist_gpop:cN`

```

5254 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
5255 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
5256 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
5257 \cs_generate_variant:Nn \clist_gpush:Nn {cN}

5258 \cs_new_protected_nopar:Npn \clist_gpop:NN {\clist_pop_aux:nnNN \tl_gset:Nn \tl_set:Nn}
5259 \cs_generate_variant:Nn \clist_gpop:NN {cN}

```

(End definition for `\clist_gpush:Nn` and others. These functions are documented on page 107.)

`\clist_top:NN` Looking at the top element of the stack without removing it is done with this operation.

`\clist_top:cN`

```

5260 \cs_new_eq:NN \clist_top:NN \clist_get:NN
5261 \cs_new_eq:NN \clist_top:cN \clist_get:cN

```

(End definition for `\clist_top:NN` and `\clist_top:cN`. These functions are documented on page 107.)

```

5262 </initex | package>

```

108 l3prop implementation

A property list is a token register whose contents is of the form

$$\backslash\mathbf{q_prop} \langle key_1 \rangle \backslash\mathbf{q_prop} \{ \langle info_1 \rangle \} \dots \backslash\mathbf{q_prop} \langle key_n \rangle \backslash\mathbf{q_prop} \{ \langle info_n \rangle \}$$

The property $\langle key \rangle$ s and $\langle info \rangle$ s might be arbitrary token lists; each $\langle info \rangle$ is surrounded by braces.

We start by ensuring that the required packages are loaded.

```

5263 \*package>
5264 \ProvidesExplPackage
5265   {\filename}{\filedate}{\fileversion}{\filedescription}
5266 \package_check_loaded_expl:
5267 </package>
5268 \*initex | package>

```

$\backslash\mathbf{q_prop}$ The separator between $\langle key \rangle$ s and $\langle info \rangle$ s and $\langle key \rangle$ s.

```

5269 \quark_new:N\q_prop

```

(End definition for $\mathbf{q_prop}$. This function is documented on page 111.)

To get values from property-lists, token lists should be passed to the appropriate functions.

108.1 Functions

$\backslash\mathbf{prop_new:N}$ Property lists are implemented as token registers.

$\backslash\mathbf{prop_new:c}$

```

5270 \cs_new_eq:NN \prop_new:N \toks_new:N
5271 \cs_new_eq:NN \prop_new:c \toks_new:c

```

(End definition for $\backslash\mathbf{prop_new:N}$. This function is documented on page 108.)

$\backslash\mathbf{prop_clear:N}$ The same goes for clearing a property list, either locally or globally.

$\backslash\mathbf{prop_clear:c}$

$\backslash\mathbf{prop_gclear:N}$

$\backslash\mathbf{prop_gclear:c}$

```

5272 \cs_new_eq:NN \prop_clear:N \toks_clear:N
5273 \cs_new_eq:NN \prop_clear:c \toks_clear:c
5274 \cs_new_eq:NN \prop_gclear:N \toks_gclear:N
5275 \cs_new_eq:NN \prop_gclear:c \toks_gclear:c

```

(End definition for $\backslash\mathbf{prop_clear:N}$. This function is documented on page 108.)

$\backslash\mathbf{prop_set_eq:NN}$ This makes two $\langle prop \rangle$ s have the same contents.

$\backslash\mathbf{prop_set_eq:Nc}$

$\backslash\mathbf{prop_set_eq:cN}$

$\backslash\mathbf{prop_set_eq:cc}$

$\backslash\mathbf{prop_gset_eq:NN}$

$\backslash\mathbf{prop_gset_eq:Nc}$

$\backslash\mathbf{prop_gset_eq:cN}$

$\backslash\mathbf{prop_gset_eq:cc}$

```

5276 \cs_new_eq:NN \prop_set_eq:NN \toks_set_eq:NN
5277 \cs_new_eq:NN \prop_set_eq:Nc \toks_set_eq:Nc
5278 \cs_new_eq:NN \prop_set_eq:cN \toks_set_eq:cN

```



```

5279 \cs_new_eq:NN \prop_set_eq:cc \toks_set_eq:cc
5280 \cs_new_eq:NN \prop_gset_eq:NN \toks_gset_eq:NN
5281 \cs_new_eq:NN \prop_gset_eq:Nc \toks_gset_eq:Nc
5282 \cs_new_eq:NN \prop_gset_eq:cN \toks_gset_eq:cN
5283 \cs_new_eq:NN \prop_gset_eq:cc \toks_gset_eq:cc

```

(End definition for `\prop_set_eq:NN`. This function is documented on page 109.)

`\prop_show:N` Show on the console the raw contents of a property list's token register.

`\prop_show:c`

```

5284 \cs_new_eq:NN \prop_show:N \toks_show:N
5285 \cs_new_eq:NN \prop_show:c \toks_show:c

```

(End definition for `\prop_show:N`. This function is documented on page 110.)

`\prop_display:N` Pretty print the contents of a property list on the console.

`\prop_display:c`

```

5286 \cs_new_protected_nopar:Npn \prop_display:N #1 {
5287   \iow_term:x { Property-list~\token_to_str:N #1~contains~
5288               the~pairs~(without~outer~braces): }
5289   \toks_clear:N \l_tmpa_toks
5290   \prop_map_inline:Nn #1 {
5291     \toks_if_empty:NF \l_tmpa_toks {
5292       \toks_put_right:Nx \l_tmpa_toks {^^J>~}
5293     }
5294     \toks_put_right:Nx \l_tmpa_toks {
5295       \c_space_tl \iow_char:N \{ \exp_not:n {##1} \iow_char:N \} \c_space_tl
5296       \c_space_tl => \c_space_tl
5297       \c_space_tl \iow_char:N \{ \exp_not:n {##2} \iow_char:N \}
5298     }
5299   }
5300   \toks_show:N \l_tmpa_toks
5301 }
5302 \cs_generate_variant:Nn \prop_display:N {c}

```

(End definition for `\prop_display:N`. This function is documented on page 110.)

`\prop_split_aux:Nnn` `\prop_split_aux:Nnn⟨prop⟩⟨key⟩⟨cmd⟩` invokes `⟨cmd⟩` with 3 arguments: 1st is the beginning of `⟨prop⟩` before `⟨key⟩`, 2nd is the value associated with `⟨key⟩`, 3rd is the rest of `⟨prop⟩` after `⟨key⟩`. If there is no property `⟨key⟩` in `⟨prop⟩`, then the 2nd argument will be `\q_no_value` and the 3rd argument is empty; otherwise the 3rd argument has the extra tokens `\q_prop ⟨key⟩ \q_prop \q_no_value` at the end.

```

5303 \cs_new_protected:Npn \prop_split_aux:Nnn #1#2#3{
5304   \cs_set:Npn \prop_tmp:w ##1 \q_prop #2 \q_prop ##2##3 \q_stop {
5305     #3 {##1}{##2}{##3}
5306   }
5307   \exp_after:wN \prop_tmp:w \toks_use:N #1 \q_prop #2 \q_prop \q_no_value \q_stop
5308 }

```

(End definition for `\prop_split_aux:Nnn`. This function is documented on page 111.)

`\prop_get:NnN` `\prop_get:NnN` $\langle prop \rangle \langle key \rangle \langle tl\ var. \rangle$ defines $\langle tl\ var. \rangle$ to be the value associated with $\langle key \rangle$ in $\langle prop \rangle$, `\q_no_value` if not found.
`\prop_get:NVN`
`\prop_get:cnN`
`\prop_get:cVN`
`\prop_get_aux:w`

```

5309 \cs_new_protected:Npn \prop_get:NnN #1#2 {
5310   \prop_split_aux:Nnn #1{#2}\prop_get_aux:w
5311 }
5312 \cs_new_protected:Npn \prop_get_aux:w #1#2#3#4 { \tl_set:Nn #4 {#2} }

5313 \cs_generate_variant:Nn \prop_get:NnN { NVN, cnN, cVN }

```

(End definition for `\prop_get:NnN`. This function is documented on page 111.)

`\prop_gget:NnN` The global version of the previous function.
`\prop_gget:NVN`
`\prop_gget:NnN`
`\prop_gget:NVN`
`\prop_gget_aux:w`

```

5314 \cs_new_protected:Npn \prop_gget:NnN #1#2{
5315   \prop_split_aux:Nnn #1{#2}\prop_gget_aux:w}
5316 \cs_new_protected:Npn \prop_gget_aux:w #1#2#3#4{\tl_gset:Nx#4{\exp_not:n{#2}}}}

5317 \cs_generate_variant:Nn \prop_gget:NnN { NVN, cnN, cVN }

```

(End definition for `\prop_gget:NnN`. This function is documented on page 111.)

`\prop_get_gdel:NnN` `\prop_get_gdel:NnN` is the same as `\prop_get:NnN` but the $\langle key \rangle$ and its value are afterwards globally removed from $\langle property_list \rangle$. One probably also needs the local variants or only the local one, or... We decide this later.
`\prop_get_del_aux:w`

```

5318 \cs_new_protected:Npn \prop_get_gdel:NnN #1#2#3{
5319   \prop_split_aux:Nnn #1{#2}{\prop_get_del_aux:w #3{\toks_gset:Nn #1{#2}}}}
5320 \cs_new_protected:Npn \prop_get_del_aux:w #1#2#3#4#5#6{
5321   \tl_set:Nn #1 {#5}
5322   \quark_if_no_value:NF #1 {
5323     \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#3\q_prop\q_no_value {#2{#4##1}}
5324     \prop_tmp:w #6}
5325 }

```

(End definition for `\prop_get_gdel:NnN`. This function is documented on page 111.)

`\prop_put:Nnn` `\prop_put:Nnn` $\{\langle prop \rangle\} \{\langle key \rangle\} \{\langle info \rangle\}$ adds/changes the value associated with $\langle key \rangle$ in $\langle prop \rangle$ to $\langle info \rangle$.
`\prop_put:Nno`
`\prop_put:NnV`
`\prop_put:Nnx`
`\prop_put:NVn`
`\prop_put:NVV`
`\prop_put:cnn`
`\prop_put:cnx`
`\prop_gput:Nnn`
`\prop_gput:NVn`
`\prop_gput:NnV`
`\prop_gput:Nno`
`\prop_gput:Nnx`
`\prop_gput:cnn`
`\prop_gput:ccx`
`\prop_put_aux:w`

```

5326 \cs_new_protected:Npn \prop_put:Nnn #1#2{
5327   \prop_split_aux:Nnn #1{#2}{
5328     \prop_clear:N #1
5329     \prop_put_aux:w {\toks_put_right:Nn #1{#2}
5330   }
5331 }

5332 \cs_new_protected:Npn \prop_gput:Nnn #1#2{
5333   \prop_split_aux:Nnn #1{#2}{
5334     \prop_gclear:N #1
5335     \prop_put_aux:w {\toks_gput_right:Nn #1{#2}
5336   }
5337 }

```

```

5338 \cs_new_protected:Npn \prop_put_aux:w #1#2#3#4#5#6{
5339   #1{\q_prop#2\q_prop{#6}#3}
5340   \tl_if_empty:nF{#5}
5341   {
5342     \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{##1}}
5343     \prop_tmp:w #5
5344   }
5345 }

5346 \cs_generate_variant:Nn \prop_put:Nnn { Nno , NnV, Nnx, NVn, NVV, cnn , cnx }

5347 \cs_generate_variant:Nn \prop_gput:Nnn { NVn,NnV,Nno,Nnx,Nox,cnn,ccx}

```

(End definition for `\prop_put:Nnn`. This function is documented on page 111.)

```

\prop_del:Nn \prop_del:Nn <prop>\<key> deletes the entry for <key> in <prop>, if any.
\prop_del:NV
\prop_gdel:NV
\prop_gdel:Nn
\prop_del_aux:w
5348 \cs_new_protected:Npn \prop_del:Nn #1#2{
5349   \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_set:Nn #1}{#2}}}
5350 \cs_new_protected:Npn \prop_gdel:Nn #1#2{
5351   \prop_split_aux:Nnn #1{#2}{\prop_del_aux:w {\toks_gset:Nn #1}{#2}}}
5352 \cs_new_protected:Npn \prop_del_aux:w #1#2#3#4#5{
5353   \cs_set_nopar:Npn \prop_tmp:w {#4}
5354   \quark_if_no_value:NF \prop_tmp:w {
5355     \cs_set_nopar:Npn \prop_tmp:w ##1\q_prop#2\q_prop\q_no_value {#1{#3##1}}
5356     \prop_tmp:w #5
5357   }
5358 }
5359 \cs_generate_variant:Nn \prop_del:Nn { NV }
5360 \cs_generate_variant:Nn \prop_gdel:Nn { NV }
5361 %

```

(End definition for `\prop_del:Nn`. This function is documented on page 111.)

```

\prop_gput_if_new:Nnn \prop_gput_if_new:Nnn <prop> <key> <info> is equivalent to
\prop_put_if_new:Nnn \prop_put_if_new:Nnn <prop> <key> <info>
\prop_if_in:NnTF <prop>\<key>
  {}%
  {\prop_gput:Nnn
    <property_list>
    <key>
    <info>}}

```

Here we go (listening to Porgy & Bess in a recording with Ella F. and Louis A. which makes writing macros sometimes difficult; I find myself humming instead of working):

```

5362 \cs_new_protected:Npn \prop_gput_if_new:Nnn #1#2{
5363   \prop_split_aux:Nnn #1{#2}{\prop_put_if_new_aux:w #1{#2}}}
5364 \cs_new_protected:Npn \prop_put_if_new_aux:w #1#2#3#4#5#6{
5365   \tl_if_empty:nT {#5}{#1{\q_prop#2\q_prop{#6}#3}}}

```

(End definition for `\prop_gput_if_new:Nnn`. This function is documented on page 111.)

108.2 Predicates and conditionals

`\prop_if_empty_p:N`
`\prop_if_empty_p:c`
`\prop_if_empty:NTF`
`\prop_if_empty:cTF`

This conditional takes a $\langle prop \rangle$ as its argument and evaluates either the true or the false case, depending on whether or not $\langle prop \rangle$ contains any properties.

5366 `\prg_new_eq_conditional:NNn \prop_if_empty:N \toks_if_empty:N {p,TF,T,F}`
5367 `\prg_new_eq_conditional:NNn \prop_if_empty:c \toks_if_empty:c {p,TF,T,F}`

(End definition for `\prop_if_empty_p:N` and `\prop_if_empty_p:c`. These functions are documented on page 110.)

`\prop_if_eq_p:NN`
`\prop_if_eq_p:cN`
`\prop_if_eq_p:Nc`
`\prop_if_eq_p:cc`
`\prop_if_eq:NNTF`
`\prop_if_eq:NcTF`
`\prop_if_eq:cNTF`
`\prop_if_eq:ccTF`

These functions test whether two property lists are equal.

5368 `\prg_new_eq_conditional:NNn \prop_if_eq:NN \toks_if_eq:NN {p,TF,T,F}`
5369 `\prg_new_eq_conditional:NNn \prop_if_eq:cN \toks_if_eq:cN {p,TF,T,F}`
5370 `\prg_new_eq_conditional:NNn \prop_if_eq:Nc \toks_if_eq:Nc {p,TF,T,F}`
5371 `\prg_new_eq_conditional:NNn \prop_if_eq:cc \toks_if_eq:cc {p,TF,T,F}`

(End definition for `\prop_if_eq_p:NN` and others. These functions are documented on page 111.)

`\prop_if_in:NnTF`
`\prop_if_in:NNTF`
`\prop_if_in:NoTF`
`\prop_if_in:cnTF`
`\prop_if_in:ccTF`
`\prop_if_in_aux:w`

`\prop_if_in:NnTF` $\langle property_list \rangle$ $\langle key \rangle$ $\langle true_case \rangle$ $\langle false_case \rangle$ will check whether or not $\langle key \rangle$ is on the $\langle property_list \rangle$ and then select either the true or false case.

5372 `\prg_new_protected_conditional:Nnn \prop_if_in:Nn {TF,T,F} {`
5373 `\prop_split_aux:Nnn #1 {#2} {\prop_if_in_aux:w}`
5374 `}`
5375 `\cs_new_nopar:Npn \prop_if_in_aux:w #1#2#3 {`
5376 `\quark_if_no_value:nTF {#2} {\prg_return_false:} {\prg_return_true:}`
5377 `}`
5378 `\cs_generate_variant:Nn \prop_if_in:NnTF {NV,No,cn,cc}`
5379 `\cs_generate_variant:Nn \prop_if_in:NnT {NV,No,cn,cc}`
5380 `\cs_generate_variant:Nn \prop_if_in:NnF {NV,No,cn,cc}`

(End definition for `\prop_if_in:Nn`. This function is documented on page 111.)

108.3 Mapping functions

`\prop_map_function:NN`
`\prop_map_function:cN`
`\prop_map_function:Nc`
`\prop_map_function:cc`
`\prop_map_function_aux:w`

Maps a function on every entry in the property list. The function must take 2 arguments: a key and a value.

First, some failed attempts:

`\cs_new_nopar:Npn \prop_map_function:NN #1#2{`
`\exp_after:wN \prop_map_function_aux:w`
`\exp_after:wN #2 \toks_use:N #1 \q_prop{} \q_prop \q_no_value \q_stop`
`}`
`\cs_new_nopar:Npn \prop_map_function_aux:w #1\q_prop#2\q_prop#3{`
`\if_predicate:w \tl_if_empty_p:n{#2}`

```

        \exp_after:wN \prop_map_break:
      \fi:
      #1{#2}{#3}
      \prop_map_function_aux:w #1
    }

```

problem with the above implementation is that an empty key stops the mapping but all other functions in the module allow the use of empty keys (as one value)

```

\cs_set_nopar:Npn \prop_map_function:NN #1#2{
  \exp_after:wN \prop_map_function_aux:w
  \exp_after:wN #2 \toks_use:N #1 \q_prop \q_no_value \q_prop \q_no_value
}
\cs_set_nopar:Npn \prop_map_function_aux:w #1\q_prop#2\q_prop#3{
  \quark_if_no_value:nF{#2}
  {
    #1{#2}{#3}
    \prop_map_function_aux:w #1
  }
}

```

problem with the above implementation is that `\quark_if_no_value:nF` is fairly slow and if `\quark_if_no_value:NF` is used instead we have to do an assignment thus making the mapping not expandable (is that important?)

Here's the current version of the code:

```

5381 \cs_set_nopar:Npn \prop_map_function:NN #1#2 {
5382   \exp_after:wN \prop_map_function_aux:w
5383   \exp_after:wN #2 \toks_use:N #1 \q_prop \q_nil \q_prop \q_no_value \q_stop
5384 }
5385 \cs_set:Npn \prop_map_function_aux:w #1 \q_prop #2 \q_prop #3 {
5386   \if_meaning:w \q_nil #2
5387   \exp_after:wN \prop_map_break:
5388   \fi:
5389   #1{#2}{#3}
5390   \prop_map_function_aux:w #1
5391 }

```

(potential) problem with the above implementation is that it will return true if #2 contains more than just `\q_nil` thus executing whatever follows. Claim: this can't happen :-)
so we should be ok

```

5392 \cs_generate_variant:Nn \prop_map_function:NN {c,Nc,cc}

```

(End definition for `\prop_map_function:NN`. This function is documented on page [111](#).)

`\prop_map_inline:Nn` The inline functions are straight forward. It takes longer to test if the list is empty than
`\prop_map_inline:cn` to run it on an empty list so we don't waste time doing that.
`\g_prop_inline_level_int`

```

5393 \int_new:N \g_prop_inline_level_int
5394 \cs_new_protected_nopar:Npn \prop_map_inline:Nn #1#2 {
5395   \int_gincr:N \g_prop_inline_level_int
5396   \cs_gset:cpn {prop_map_inline_ \int_use:N \g_prop_inline_level_int :n}
5397     ##1##2{#2}
5398   \prop_map_function:Nc #1
5399     {prop_map_inline_ \int_use:N \g_prop_inline_level_int :n}
5400   \int_gdecr:N \g_prop_inline_level_int
5401 }

```

```

5402 \cs_generate_variant:Nn\prop_map_inline:Nn {cn}

```

(End definition for `\prop_map_inline:Nn`. This function is documented on page 111.)

`\prop_map_break:` The break statement.

```

5403 \cs_new_eq:NN \prop_map_break: \use_none_delimit_by_q_stop:w

```

(End definition for `\prop_map_break:.` This function is documented on page 110.)

```

5404 </initex | package>

```

109 l3box implementation

Announce and ensure that the required packages are loaded.

```

5405 <*package>
5406 \ProvidesExplPackage
5407   {\filename}{\filedate}{\fileversion}{\filedescription}
5408   \package_check_loaded_expl:
5409 </package>
5410 <*initex | package>

```

The code in this module is very straight forward so I'm not going to comment it very extensively.

109.1 Generic boxes

`\box_new:N` Defining a new `<box>` register.

`\box_new:c` (The test suite for this command, and others in this file, is `m3box001.lvt`.)

```

5411 <*initex>
5412 \alloc_new:nnnN {box} \c_zero \c_max_register_int \tex_mathchardef:D

```

Now, remember that `\box255` has a special role in \TeX , it shouldn't be allocated...

```
5413 \seq_put_right:Nn \g_box_allocation_seq {255}
5414 \initex
```

When we run on top of \LaTeX , we just use its allocation mechanism.

```
5415 \*package>
5416 \cs_new_protected:Npn \box_new:N #1 {
5417   \chk_if_free_cs:N #1
5418   \newbox #1
5419 }
5420 \package>

5421 \cs_generate_variant:Nn \box_new:N {c}
```

(End definition for `\box_new:N` and `\box_new:c`. These functions are documented on page 112.)

```
\if_hbox:N
\if_vbox:N
\if_box_empty:N
```

The primitives for testing if a $\langle box \rangle$ is empty/void or which type of box it is.

```
5422 \cs_new_eq:NN \if_hbox:N \tex_ifhbox:D
5423 \cs_new_eq:NN \if_vbox:N \tex_ifvbox:D
5424 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
```

(End definition for `\if_hbox:N`, `\if_vbox:N`, and `\if_box_empty:N`. These functions are documented on page 112.)

```
\box_if_horizontal_p:N
\box_if_horizontal_p:c
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_horizontal:NTF
\box_if_horizontal:cTF
\box_if_vertical:NTF
\box_if_vertical:cTF
```

```
5425 \prg_new_conditional:Nnn \box_if_horizontal:N {p,TF,T,F} {
5426   \tex_ifhbox:D #1 \prg_return_true: \else: \prg_return_false: \fi:
5427 }
5428 \prg_new_conditional:Nnn \box_if_vertical:N {p,TF,T,F} {
5429   \tex_ifvbox:D #1 \prg_return_true: \else: \prg_return_false: \fi:
5430 }
5431 \cs_generate_variant:Nn \box_if_horizontal_p:N {c}
5432 \cs_generate_variant:Nn \box_if_horizontal:N {NTF} {c}
5433 \cs_generate_variant:Nn \box_if_horizontal:NT {c}
5434 \cs_generate_variant:Nn \box_if_horizontal:NF {c}
5435 \cs_generate_variant:Nn \box_if_vertical_p:N {c}
5436 \cs_generate_variant:Nn \box_if_vertical:NTF {c}
5437 \cs_generate_variant:Nn \box_if_vertical:NT {c}
5438 \cs_generate_variant:Nn \box_if_vertical:NF {c}
```

(End definition for `\box_if_horizontal_p:N` and `\box_if_horizontal_p:c`. These functions are documented on page 113.)

```
\box_if_empty_p:N
\box_if_empty_p:c
\box_if_empty:NTF
\box_if_empty:cTF
```

Testing if a $\langle box \rangle$ is empty/void.

```
5439 \prg_new_conditional:Nnn \box_if_empty:N {p,TF,T,F} {
5440   \tex_ifvoid:D #1 \prg_return_true: \else: \prg_return_false: \fi:
```

```

5441 }
5442 \cs_generate_variant:Nn \box_if_empty_p:N {c}
5443 \cs_generate_variant:Nn \box_if_empty:NTF {c}
5444 \cs_generate_variant:Nn \box_if_empty:NT {c}
5445 \cs_generate_variant:Nn \box_if_empty:NF {c}

```

(End definition for `\box_if_empty_p:N` and `\box_if_empty_p:c`. These functions are documented on page 113.)

`\box_set_eq:NN` Assigning the contents of a box to be another box.

```

\box_set_eq:cN
\box_set_eq:Nc
\box_set_eq:cc
5446 \cs_new_protected_nopar:Npn \box_set_eq:NN #1#2 {\tex_setbox:D #1 \tex_copy:D #2}
5447 \cs_generate_variant:Nn \box_set_eq:NN {cN,Nc,cc}

```

(End definition for `\box_set_eq:NN` and others. These functions are documented on page 113.)

`\box_set_eq_clear:NN` Assigning the contents of a box to be another box. This clears the second box globally (that's how \TeX does it).

```

\box_set_eq_clear:cN
\box_set_eq_clear:Nc
\box_set_eq_clear:cc
5448 \cs_new_protected_nopar:Npn \box_set_eq_clear:NN #1#2 {\tex_setbox:D #1 \tex_box:D #2}
5449 \cs_generate_variant:Nn \box_set_eq_clear:NN {cN,Nc,cc}

```

(End definition for `\box_set_eq_clear:NN` and others. These functions are documented on page 113.)

`\box_gset_eq:NN` Global version of the above.

```

\box_gset_eq:cN
\box_gset_eq:Nc
\box_gset_eq:cc
\box_gset_eq_clear:NN
\box_gset_eq_clear:cN
\box_gset_eq_clear:Nc
\box_gset_eq_clear:cc
5450 \cs_new_protected_nopar:Npn \box_gset_eq:NN {\pref_global:D\box_set_eq:NN}
5451 \cs_generate_variant:Nn \box_gset_eq:NN {cN,Nc,cc}
5452 \cs_new_protected_nopar:Npn \box_gset_eq_clear:NN {\pref_global:D\box_set_eq_clear:NN}
5453 \cs_generate_variant:Nn \box_gset_eq_clear:NN {cN,Nc,cc}

```

(End definition for `\box_gset_eq:NN` and others. These functions are documented on page 113.)

`\l_last_box` A different name for this read-only primitive.

```

5454 \cs_new_eq:NN \l_last_box \tex_lastbox:D

```

`\box_set_to_last:N` Set a box to the previous box.

```

\box_gset_to_last:N
\box_set_to_last:c
\box_gset_to_last:c
5455 \cs_new_protected_nopar:Npn \box_set_to_last:N #1{\tex_setbox:D#1\l_last_box}
5456 \cs_generate_variant:Nn \box_set_to_last:N {c}
5457 \cs_new_protected_nopar:Npn \box_gset_to_last:N {\pref_global:D \box_set_to_last:N}
5458 \cs_generate_variant:Nn \box_gset_to_last:N {c}

```

(End definition for `\box_set_to_last:N` and others. These functions are documented on page 114.)

`\box_move_left:nn` Move box material in different directions.

```

\box_move_right:nn
\box_move_up:nn
\box_move_down:nn
5459 \cs_new:Npn \box_move_left:nn #1#2{\tex_moveleft:D\dim_eval:n{#1} #2}
5460 \cs_new:Npn \box_move_right:nn #1#2{\tex_moveright:D\dim_eval:n{#1} #2}
5461 \cs_new:Npn \box_move_up:nn #1#2{\tex_raise:D\dim_eval:n{#1} #2}
5462 \cs_new:Npn \box_move_down:nn #1#2{\tex_lower:D\dim_eval:n{#1} #2}

```


(End definition for `\box_move_left:n` and others. These functions are documented on page 114.)

`\box_clear:N` Clear a $\langle box \rangle$ register.
`\box_clear:c`
`\box_gclear:N` 5463 `\cs_new_protected_nopar:Npn \box_clear:N #1{\box_set_eq_clear:NN #1 \c_empty_box }`
`\box_gclear:c` 5464 `\cs_generate_variant:Nn \box_clear:N {c}`
5465 `\cs_new_protected_nopar:Npn \box_gclear:N {\pref_global:D\box_clear:N}`
5466 `\cs_generate_variant:Nn \box_gclear:N {c}`

(End definition for `\box_clear:N` and others. These functions are documented on page 114.)

`\box_ht:N` Accessing the height, depth, and width of a $\langle box \rangle$ register.
`\box_ht:c`
`\box_dp:N` 5467 `\cs_new_eq:NN \box_ht:N \tex_ht:D`
`\box_dp:c` 5468 `\cs_new_eq:NN \box_dp:N \tex_dp:D`
`\box_wd:N` 5469 `\cs_new_eq:NN \box_wd:N \tex_wd:D`
`\box_wd:c` 5470 `\cs_generate_variant:Nn \box_ht:N {c}`
5471 `\cs_generate_variant:Nn \box_dp:N {c}`
5472 `\cs_generate_variant:Nn \box_wd:N {c}`

(End definition for `\box_ht:N` and others. These functions are documented on page 114.)

`\box_set_ht:Nn` Measuring is easy: all primitive work. These primitives are not expandable, so the derived
`\box_set_ht:cn` functions are not either.
`\box_set_dp:Nn` 5473 `\cs_new_protected_nopar:Npn \box_set_dp:Nn #1#2 {`
`\box_set_dp:cn` 5474 `\box_dp:N #1 \etex_dimexpr:D #2 \scan_stop:`
`\box_set_wd:Nn` 5475 `}`
`\box_set_wd:cn` 5476 `\cs_new_protected_nopar:Npn \box_set_ht:Nn #1#2 {`
5477 `\box_ht:N #1 \etex_dimexpr:D #2 \scan_stop:`
5478 `}`
5479 `\cs_new_protected_nopar:Npn \box_set_wd:Nn #1#2 {`
5480 `\box_wd:N #1 \etex_dimexpr:D #2 \scan_stop:`
5481 `}`
5482 `\cs_generate_variant:Nn \box_set_ht:Nn { c }`
5483 `\cs_generate_variant:Nn \box_set_dp:Nn { c }`
5484 `\cs_generate_variant:Nn \box_set_wd:Nn { c }`

(End definition for `\box_set_ht:Nn` and others. These functions are documented on page 115.)

`\box_use_clear:N` Using a $\langle box \rangle$. These are just TeX primitives with meaningful names.
`\box_use_clear:c`
`\box_use:N` 5485 `\cs_new_eq:NN \box_use_clear:N \tex_box:D`
`\box_use:c` 5486 `\cs_generate_variant:Nn \box_use_clear:N {c}`
5487 `\cs_new_eq:NN \box_use:N \tex_copy:D`
5488 `\cs_generate_variant:Nn \box_use:N {c}`

(End definition for `\box_use_clear:N` and others. These functions are documented on page 114.)

`\box_show:N` Show the contents of a box and write it into the log file.

`\box_show:c`

```
5489 \cs_set_eq:NN \box_show:N \tex_showbox:D
5490 \cs_generate_variant:Nn \box_show:N {c}
```

(End definition for `\box_show:N` and `\box_show:c`. These functions are documented on page 115.)

`\c_empty_box` We allocate some `\box` registers here (and borrow a few from L^AT_EX).

`\l_tmpa_box`

`\l_tmpb_box`

```
5491 <package>\cs_set_eq:NN \c_empty_box \voidb@x
5492 <package>\cs_new_eq:NN \l_tmpa_box \@tempboxa
5493 <initex>\box_new:N \c_empty_box
5494 <initex>\box_new:N \l_tmpa_box
5495 \box_new:N \l_tmpb_box
```

109.2 Vertical boxes

`\vbox:n` (The test suite for this command, and others in this file, is `m3box003.lvt`.)

`\vbox_top:n`

Put a vertical box directly into the input stream.

```
5496 \cs_new_protected_nopar:Npn \vbox:n { \tex_vbox:D \scan_stop: }
5497 \cs_new_protected_nopar:Npn \vbox_top:n { \tex_vtop:D \scan_stop: }
```

(End definition for `\vbox:n` and `\vbox_top:n`. These functions are documented on page 117.)

`\vbox_set:Nn` Storing material in a vertical box with a natural height.

`\vbox_set:cn`

`\vbox_gset:Nn`

`\vbox_gset:cn`

```
5498 \cs_new_protected:Npn \vbox_set:Nn #1#2 { \tex_setbox:D #1 \tex_vbox:D {#2} }
5499 \cs_generate_variant:Nn \vbox_set:Nn {cn}
5500 \cs_new_protected_nopar:Npn \vbox_gset:Nn { \pref_global:D \vbox_set:Nn }
5501 \cs_generate_variant:Nn \vbox_gset:Nn {cn}
```

(End definition for `\vbox_set:Nn` and others. These functions are documented on page 117.)

`\vbox_set_top:Nn` Storing material in a vertical box with a natural height and reference point at the baseline
`\vbox_set_top:cn` of the first object in the box.

`\vbox_gset_top:Nn`

`\vbox_gset_top:cn`

```
5502 \cs_new_protected:Npn \vbox_set_top:Nn #1#2 { \tex_setbox:D #1 \tex_vtop:D {#2} }
5503 \cs_generate_variant:Nn \vbox_set_top:Nn {cn}
5504 \cs_new_protected_nopar:Npn \vbox_gset_top:Nn { \pref_global:D \vbox_set_top:Nn }
5505 \cs_generate_variant:Nn \vbox_gset_top:Nn {cn}
```

(End definition for `\vbox_set_top:Nn` and others. These functions are documented on page 117.)

`\vbox_set_to_ht:Nnn` Storing material in a vertical box with a specified height.

`\vbox_set_to_ht:cnn`

`\vbox_gset_to_ht:Nnn`

`\vbox_gset_to_ht:cnn`

`\vbox_gset_to_ht:ccn`

```
5506 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3 {
5507   \tex_setbox:D #1 \tex_vbox:D to #2 {#3}
5508 }
5509 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn {cnn}
5510 \cs_new_protected_nopar:Npn \vbox_gset_to_ht:Nnn { \pref_global:D \vbox_set_to_ht:Nnn }
5511 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn {cnn,ccn}
```

(End definition for `\vbox_set_to_ht:Nnn` and others. These functions are documented on page 118.)

`\vbox_set_inline_begin:N` Storing material in a vertical box. This type is useful in environment definitions.
`\vbox_set_inline_end:`
`\vbox_gset_inline_begin:N`
`\vbox_gset_inline_end:`

```
5512 \cs_new_protected_nopar:Npn \vbox_set_inline_begin:N #1 {
5513   \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }
5514 \cs_new_eq:NN \vbox_set_inline_end: \c_group_end_token
5515 \cs_new_protected_nopar:Npn \vbox_gset_inline_begin:N {
5516   \pref_global:D \vbox_set_inline_begin:N }
5517 \cs_new_eq:NN \vbox_gset_inline_end: \c_group_end_token
```

(End definition for `\vbox_set_inline_begin:N` and others. These functions are documented on page 118.)

`\vbox_to_ht:nn` Put a vertical box directly into the input stream.
`\vbox_to_zero:n`

```
5518 \cs_new_protected:Npn \vbox_to_ht:nn #1#2{\tex_vbox:D to \dim_eval:n{#1}{#2}}
5519 \cs_new_protected:Npn \vbox_to_zero:n #1 {\tex_vbox:D to \c_zero_dim {#1}}
```

(End definition for `\vbox_to_ht:nn` and `\vbox_to_zero:n`. These functions are documented on page 118.)

`\vbox_set_split_to_ht:NNn` Splitting a vertical box in two.

```
5520 \cs_new_protected_nopar:Npn \vbox_set_split_to_ht:NNn #1#2#3{
5521   \tex_setbox:D #1 \tex_vsplit:D #2 to #3
5522 }
```

(End definition for `\vbox_set_split_to_ht:NNn`. This function is documented on page 118.)

`\vbox_unpack:N` Unpacking a box and if requested also clear it.
`\vbox_unpack:c`
`\vbox_unpack_clear:N`
`\vbox_unpack_clear:c`

```
5523 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
5524 \cs_generate_variant:Nn \vbox_unpack:N {c}
5525 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
5526 \cs_generate_variant:Nn \vbox_unpack_clear:N {c}
```

(End definition for `\vbox_unpack:N` and others. These functions are documented on page 118.)

109.3 Horizontal boxes

`\hbox:n` Put a horizontal box directly into the input stream.

(The test suite for this command, and others in this file, is `m3box002.lvt`.)

```
5527 \cs_new_protected_nopar:Npn \hbox:n {\tex_hbox:D \scan_stop:}
```

(End definition for `\hbox:n`. This function is documented on page 116.)

`\hbox_set:Nn` Assigning the contents of a box to be another box. This clears the second box globally (that's how TeX does it).

`\hbox_set:cn`
`\hbox_gset:Nn`
`\hbox_gset:cn`

```

5528 \cs_new_protected:Npn \hbox_set:Nn #1#2 {\tex_setbox:D #1 \tex_hbox:D {#2}}
5529 \cs_generate_variant:Nn \hbox_set:Nn {cn}
5530 \cs_new_protected_nopar:Npn \hbox_gset:Nn {\pref_global:D \hbox_set:Nn}
5531 \cs_generate_variant:Nn \hbox_gset:Nn {cn}

```

(End definition for `\hbox_set:Nn` and others. These functions are documented on page 116.)

`\hbox_set_to_wd:Nnn` Storing material in a horizontal box with a specified width.

`\hbox_set_to_wd:cnn`
`\hbox_gset_to_wd:Nnn`
`\hbox_gset_to_wd:cnn`

```

5532 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3 {
5533   \tex_setbox:D #1 \tex_hbox:D to \dim_eval:n{#2} {#3}
5534 }
5535 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn {cnn}
5536 \cs_new_protected_nopar:Npn \hbox_gset_to_wd:Nnn {\pref_global:D \hbox_set_to_wd:Nnn }
5537 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn {cnn}

```

(End definition for `\hbox_set_to_wd:Nnn` and others. These functions are documented on page 116.)

`\hbox_set_inline_begin:N` Storing material in a horizontal box. This type is useful in environment definitions.

`\hbox_set_inline_begin:c`
`\hbox_gset_inline_begin:N`
`\hbox_gset_inline_begin:c`
`\hbox_set_inline_end:`
`\hbox_gset_inline_end:`

```

5538 \cs_new_protected_nopar:Npn \hbox_set_inline_begin:N #1 {
5539   \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token
5540 }
5541 \cs_generate_variant:Nn \hbox_set_inline_begin:N {c}
5542 \cs_new_eq:NN \hbox_set_inline_end: \c_group_end_token
5543 \cs_new_protected_nopar:Npn \hbox_gset_inline_begin:N {
5544   \pref_global:D \hbox_set_inline_begin:N
5545 }
5546 \cs_generate_variant:Nn \hbox_gset_inline_begin:N {c}
5547 \cs_new_eq:NN \hbox_gset_inline_end: \c_group_end_token

```

(End definition for `\hbox_set_inline_begin:N` and others. These functions are documented on page 116.)

`\hbox_to_wd:nn` Put a horizontal box directly into the input stream.

`\hbox_to_zero:n`

```

5548 \cs_new_protected:Npn \hbox_to_wd:nn #1#2 {\tex_hbox:D to #1 {#2}}
5549 \cs_new_protected:Npn \hbox_to_zero:n #1 {\tex_hbox:D to \c_zero_skip {#1}}

```

(End definition for `\hbox_to_wd:nn` and `\hbox_to_zero:n`. These functions are documented on page 116.)

`\hbox_overlap_left:n` Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

`\hbox_overlap_right:n`

```

5550 \cs_new_protected:Npn \hbox_overlap_left:n #1 {\hbox_to_zero:n {\tex_hss:D #1}}
5551 \cs_new_protected:Npn \hbox_overlap_right:n #1 {\hbox_to_zero:n {#1 \tex_hss:D}}

```

(End definition for `\hbox_overlap_left:n` and `\hbox_overlap_right:n`. These functions are documented on page 116.)

```

\hbox_unpack:N Unpacking a box and if requested also clear it.
\hbox_unpack:c
\hbox_unpack_clear:N
\hbox_unpack_clear:c
5552 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
5553 \cs_generate_variant:Nn \hbox_unpack:N {c}
5554 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
5555 \cs_generate_variant:Nn \hbox_unpack_clear:N {c}

```

(End definition for `\hbox_unpack:N` and others. These functions are documented on page 117.)

```
5556 \</initex | package>
```

110 l3io implementation

We start by ensuring that the required packages are loaded.

```

5557 <*package>
5558 \ProvidesExplPackage
5559   {\filename}{\filedate}{\fileversion}{\filedescription}
5560   \package_check_loaded_expl:
5561 </package>
5562 <*initex | package>

```

110.1 Variables and constants

`\c_iow_term_stream` Here we allocate two output streams for writing to the transcript file only (`\c_iow_log_stream`) and to both the terminal and transcript file (`\c_iow_term_stream`). Both can be used to read from and have equivalent `\c_ior` versions.

```

5563 \cs_new_eq:NN \c_iow_term_stream \c_sixteen
5564 \cs_new_eq:NN \c_ior_term_stream \c_sixteen
5565 \cs_new_eq:NN \c_iow_log_stream \c_minus_one
5566 \cs_new_eq:NN \c_ior_log_stream \c_minus_one

```

(End definition for `\c_iow_term_stream`. This function is documented on page 123.)

`\c_iow_streams_tl` The list of streams available, by number.
`\c_ior_streams_tl`

```

5567 \tl_const:Nn \c_iow_streams_tl
5568 {
5569   \c_zero
5570   \c_one
5571   \c_two
5572   \c_three
5573   \c_four

```

```

5574 \c_five
5575 \c_six
5576 \c_seven
5577 \c_eight
5578 \c_nine
5579 \c_ten
5580 \c_eleven
5581 \c_twelve
5582 \c_thirteen
5583 \c_fourteen
5584 \c_fifteen
5585 }
5586 \cs_new_eq:NN \c_ior_streams_tl \c_iow_streams_tl

```

(End definition for `\c_iow_streams_tl`. This function is documented on page ??.)

`\g_iow_streams_prop` The allocations for streams are stored in property lists, which are set up to have a 'full'
`\g_ior_streams_prop` set of allocations from the start. In package mode, a few slots are always taken, so these
are blocked off from use.

```

5587 \prop_new:N \g_iow_streams_prop
5588 \prop_new:N \g_ior_streams_prop
5589 </initex | package>
5590 <*package>
5591 \prop_put:Nnn \g_iow_streams_prop { 0 } { LaTeX2e~reserved }
5592 \prop_put:Nnn \g_iow_streams_prop { 1 } { LaTeX2e~reserved }
5593 \prop_put:Nnn \g_iow_streams_prop { 2 } { LaTeX2e~reserved }
5594 \prop_put:Nnn \g_ior_streams_prop { 0 } { LaTeX2e~reserved }
5595 </package>
5596 <*initex | package>

```

(End definition for `\g_iow_streams_prop`. This function is documented on page 123.)

`\l_iow_stream_int` Used to track the number allocated to the stream being created: this is taken from the
`\l_ior_stream_int` property list but does alter.

```

5597 \int_new:N \l_iow_stream_int
5598 \cs_new_eq:NN \l_ior_stream_int \l_iow_stream_int

```

(End definition for `\l_iow_stream_int`. This function is documented on page 123.)

110.2 Stream management

`\iow_raw_new:N` The lowest level for stream management is actually creating raw T_EX streams. As these
`\ior_raw_new:N` are very limited (even with ϵ -T_EX) this should not be addressed directly.

```

5599 </initex | package>
5600 <*initex>
5601 \alloc_setup_type:nnn { iow } \c_zero \c_sixteen

```

```

5602 \cs_new_protected_nopar:Npn \iow_raw_new:N #1 {
5603   \alloc_reg:NnNN g { iow } \tex_chardef:D #1
5604 }
5605 \alloc_setup_type:nnn { ior } \c_zero \c_sixteen
5606 \cs_new_protected_nopar:Npn \ior_raw_new:N #1 {
5607   \alloc_reg:NnNN g { ior } \tex_chardef:D #1
5608 }
5609 </initex>
5610 <*package>
5611 \cs_set_eq:NN \iow_raw_new:N \newwrite
5612 \cs_set_eq:NN \ior_raw_new:N \newread
5613 </package>
5614 <*initex | package>
5615 \cs_generate_variant:Nn \iow_raw_new:N { c }
5616 \cs_generate_variant:Nn \ior_raw_new:N { c }

```

(End definition for `\iow_raw_new:N`. This function is documented on page 122.)

`\iow_new:N` These are not needed but are included for consistency with other variable types.

```

\iow_new:c
\iow_new:c
\ior_new:N
\ior_new:c
5617 \cs_new_protected_nopar:Npn \iow_new:N #1 {
5618   \cs_new_eq:NN #1 \c_iow_log_stream
5619 }
5620 \cs_generate_variant:Nn \iow_new:N { c }
5621 \cs_new_protected_nopar:Npn \ior_new:N #1 {
5622   \cs_new_eq:NN #1 \c_ior_log_stream
5623 }
5624 \cs_generate_variant:Nn \ior_new:N { c }

```

(End definition for `\iow_new:N`. This function is documented on page 119.)

`\iow_open:Nn` In both cases, opening a stream starts with a call to the closing function: this is safest.
`\iow_open:cn` There is then a loop through the allocation number list to find the first free stream
`\ior_open:Nn` number. When one is found the allocation can take place, the information can be stored
`\ior_open:cn` and finally the file can actually be opened.

```

5625 \cs_new_protected_nopar:Npn \iow_open:Nn #1#2 {
5626   \iow_close:N #1
5627   \int_set:Nn \l_iow_stream_int { \c_sixteen }
5628   \tl_map_function:NN \c_iow_streams_tl \iow_alloc_write:n
5629   \int_compare:nTF { \l_iow_stream_int = \c_sixteen }
5630     { \msg_kernel_error:nn { iow } { streams-exhausted } }
5631     {
5632       \iow_stream_alloc:N #1
5633       \prop_gput:Nvn \g_iow_streams_prop \l_iow_stream_int {#2}
5634       \tex_immediate:D \tex_openout:D #1#2 \scan_stop:
5635     }
5636 }
5637 \cs_generate_variant:Nn \iow_open:Nn { c }
5638 \cs_new_protected_nopar:Npn \ior_open:Nn #1#2 {

```

```

5639 \ior_close:N #1
5640 \int_set:Nn \l_ior_stream_int { \c_sixteen }
5641 \tl_map_function:NN \c_ior_streams_tl \ior_alloc_read:n
5642 \int_compare:nTF { \l_ior_stream_int = \c_sixteen }
5643   { \msg_kernel_error:nn { ior } { streams-exhausted } }
5644   {
5645     \ior_stream_alloc:N #1
5646     \prop_gput:Nvn \g_ior_streams_prop \l_ior_stream_int {#2}
5647     \tex_openin:D #1#2 \scan_stop:
5648   }
5649 }
5650 \cs_generate_variant:Nn \ior_open:Nn { c }

```

(End definition for `\ior_open:Nn`. This function is documented on page 119.)

`\iow_alloc_write:n` These functions are used to see if a particular stream is available. The property list
`\ior_alloc_read:n` contains file names for streams in use, so any unused ones are for the taking.

```

5651 \cs_new_protected_nopar:Npn \iow_alloc_write:n #1 {
5652   \prop_if_in:NnF \g_iow_streams_prop {#1}
5653   {
5654     \int_set:Nn \l_iow_stream_int {#1}
5655     \tl_map_break:
5656   }
5657 }
5658 \cs_new_protected_nopar:Npn \ior_alloc_read:n #1 {
5659   \prop_if_in:NnF \g_iow_streams_prop {#1}
5660   {
5661     \int_set:Nn \l_ior_stream_int {#1}
5662     \tl_map_break:
5663   }
5664 }

```

(End definition for `\iow_alloc_write:n`.)

`\iow_stream_alloc:N` Allocating a raw stream is much easier in initex mode than for the package. For the
`\ior_stream_alloc:N` format, all streams will be allocated by l3io and so there is a simple check to see if a
`\iow_stream_alloc_aux:` raw stream is actually available. On the other hand, for the package there will be non-
`\ior_stream_alloc_aux:` managed streams. So if the managed one is not open, a check is made to see if some
`\g_iow_tmp_stream` other managed stream is available before deciding to open a new one. If a new one is
`\g_ior_tmp_stream` needed, we get the number allocated by L^AT_EX 2_ε to get ‘back on track’ with allocation.

```

5665 \cs_new_protected_nopar:Npn \iow_stream_alloc:N #1 {
5666   \cs_if_exist:cTF { g_iow_ \int_use:N \l_iow_stream_int _stream }
5667   { \cs_gset_eq:Nc #1 { g_iow_ \int_use:N \l_iow_stream_int _stream } }
5668   {
5669     </initex | package>
5670     <*package>
5671     \iow_stream_alloc_aux:
5672     \int_compare:nT { \l_iow_stream_int = \c_sixteen }

```



```

5673     {
5674         \iow_raw_new:N \g_iow_tmp_stream
5675         \int_set:Nn \l_iow_stream_int { \g_iow_tmp_stream }
5676         \cs_gset_eq:cN
5677         { \g_iow_ \int_use:N \l_iow_stream_int _stream }
5678         \g_iow_tmp_stream
5679     }
5680 </package>
5681 <*initex>
5682     \iow_raw_new:c { \g_iow_ \int_use:N \l_iow_stream_int _stream }
5683 </initex>
5684 <*initex|package>
5685     \cs_gset_eq:Nc #1 { \g_iow_ \int_use:N \l_iow_stream_int _stream }
5686 }
5687 }
5688 </initex|package>
5689 <*package>
5690 \cs_new_protected_nopar:Npn \iow_stream_alloc_aux: {
5691     \int_incr:N \l_iow_stream_int
5692     \int_compare:nT
5693     { \l_iow_stream_int < \c_sixteen }
5694     {
5695         \cs_if_exist:cTF { \g_iow_ \int_use:N \l_iow_stream_int _stream }
5696         {
5697             \prop_if_in:NVT \g_iow_streams_prop \l_iow_stream_int
5698             { \iow_stream_alloc_aux: }
5699         }
5700         { \iow_stream_alloc_aux: }
5701     }
5702 }
5703 </package>
5704 <*initex|package>
5705 \cs_new_protected_nopar:Npn \ior_stream_alloc:N #1 {
5706     \cs_if_exist:cTF { \g_ior_ \int_use:N \l_ior_stream_int _stream }
5707     { \cs_gset_eq:Nc #1 { \g_ior_ \int_use:N \l_ior_stream_int _stream } }
5708     {
5709 </initex|package>
5710 <*package>
5711         \ior_stream_alloc_aux:
5712         \int_compare:nT { \l_ior_stream_int = \c_sixteen }
5713         {
5714             \ior_raw_new:N \g_ior_tmp_stream
5715             \int_set:Nn \l_ior_stream_int { \g_ior_tmp_stream }
5716             \cs_gset_eq:cN
5717             { \g_ior_ \int_use:N \l_ior_stream_int _stream }
5718             \g_ior_tmp_stream
5719         }
5720 </package>
5721 <*initex>
5722     \ior_raw_new:c { \g_ior_ \int_use:N \l_ior_stream_int _stream }

```

```

5723 </initex>
5724 <*initex|package>
5725     \cs_gset_eq:Nc #1 { g_ior_ \int_use:N \l_ior_stream_int_stream }
5726 }
5727 }
5728 </initex|package>
5729 <*package>
5730 \cs_new_protected_nopar:Npn \ior_stream_alloc_aux: {
5731     \int_incr:N \l_ior_stream_int
5732     \int_compare:nT
5733         { \l_ior_stream_int < \c_sixteen }
5734     {
5735         \cs_if_exist:cTF { g_ior_ \int_use:N \l_ior_stream_int_stream }
5736         {
5737             \prop_if_in:NVT \g_ior_streams_prop \l_ior_stream_int
5738             { \ior_stream_alloc_aux: }
5739         }
5740         { \ior_stream_alloc_aux: }
5741     }
5742 }
5743 </package>
5744 <*initex|package>

```

(End definition for \ior_stream_alloc:N.)

\ior_close:N Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

\ior_close:c

\ior_close:N

\ior_close:c

```

5745 \cs_new_protected_nopar:Npn \ior_close:N #1 {
5746     \cs_if_exist:NT #1
5747     {
5748         \int_compare:nF { #1 = \c_minus_one }
5749         {
5750             \tex_immediate:D \tex_closeout:D #1
5751             \prop_gdel:N \g_iow_streams_prop #1
5752             \cs_gundefine:N #1
5753         }
5754     }
5755 }
5756 \cs_generate_variant:Nn \ior_close:N { c }
5757 \cs_new_protected_nopar:Npn \ior_close:c #1 {
5758     \cs_if_exist:NT #1
5759     {
5760         \int_compare:nF { #1 = \c_minus_one }
5761         {
5762             \tex_closein:D #1
5763             \prop_gdel:N \g_ior_streams_prop #1
5764             \cs_gundefine:N #1
5765         }
5766     }
5767 }

```

```

5766     }
5767   }
5768   \cs_generate_variant:Nn \ior_close:N { c }

```

(End definition for `\ior_close:N`. This function is documented on page 120.)

`\iow_open_streams:` Simply show the property lists.

`\ior_open_streams:`

```

5769   \cs_new_protected_nopar:Npn \iow_open_streams: {
5770     \prop_display:N \g_iow_streams_prop
5771   }
5772   \cs_new_protected_nopar:Npn \ior_open_streams: {
5773     \prop_display:N \g_ior_streams_prop
5774   }

```

(End definition for `\iow_open_streams:`. This function is documented on page ??.)

Text for the error messages.

```

5775   \msg_kernel_new:nnnn { iow } { streams-exhausted }
5776   {Output streams exhausted}
5777   {%
5778     TeX can only open up to 16 output streams at one time.\\%
5779     All 16 are currently in use, and something wanted to open
5780     another one.%
5781   }
5782   \msg_kernel_new:nnnn { ior } { streams-exhausted }
5783   {Input streams exhausted}
5784   {%
5785     TeX can only open up to 16 input streams at one time.\\%
5786     All 16 are currently in use, and something wanted to open
5787     another one.%
5788   }

```

110.3 Immediate writing

`\iow_now:Nx` An abbreviation for an often used operation, which immediately writes its second argument expanded to the output stream.

```

5789   \cs_new_protected_nopar:Npn \iow_now:Nx { \tex_immediate:D \iow_shipout_x:Nn }

```

(End definition for `\iow_now:Nx`. This function is documented on page 120.)

`\iow_now:Nn` This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error.

```

5790   \cs_new_protected_nopar:Npn \iow_now:Nn #1#2 {
5791     \iow_now:Nx #1 { \exp_not:n {#2} }
5792   }

```

(End definition for `\iow_now:Nn`. This function is documented on page 120.)

`\iow_log:n` Now we redefine two functions for which we needed a definition very early on.
`\iow_log:x`
`\iow_term:n`
`\iow_term:x`

```

5793 \cs_set_protected_nopar:Npn \iow_log:x { \iow_now:Nx \c_iow_log_stream }
5794 \cs_new_protected_nopar:Npn \iow_log:n { \iow_now:Nn \c_iow_log_stream }
5795 \cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_iow_term_stream }
5796 \cs_new_protected_nopar:Npn \iow_term:n { \iow_now:Nn \c_iow_term_stream }

```

(End definition for `\iow_log:n`. This function is documented on page 120.)

`\iow_now_when_avail:Nn` For writing only if the stream requested is open at all.

`\iow_now_when_avail:cn`
`\iow_now_when_avail:Nx`
`\iow_now_when_avail:cx`

```

5797 \cs_new_protected_nopar:Npn \iow_now_when_avail:Nn #1 {
5798   \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nn #1 }
5799 }
5800 \cs_generate_variant:Nn \iow_now_when_avail:Nn { c }
5801 \cs_new_protected_nopar:Npn \iow_now_when_avail:Nx #1 {
5802   \cs_if_free:NTF #1 { \use_none:n } { \iow_now:Nx #1 }
5803 }
5804 \cs_generate_variant:Nn \iow_now_when_avail:Nx { c }

```

(End definition for `\iow_now_when_avail:Nn`. This function is documented on page 121.)

`\iow_now_buffer_safe:Nn` Another type of writing onto an output stream is used for potentially long token sequences. We break the output lines at every blank in the second argument. This avoids the problem of buffer overflow when reading back, or badly broken lines on systems with limited file records. The only thing we have to take care of, is the danger of two blanks in succession since these get converted into a `\par` when we read the stuff back. But this can happen only if things like two spaces find their way into the second argument. Usually, multiple spaces are removed by TeX's scanner.

`\iow_now_buffer_safe:Nx`
`\iow_now_buffer_safe_expanded_aux:w`

```

5805 \cs_new_protected_nopar:Npn \iow_now_buffer_safe:Nn {
5806   \iow_now_buffer_safe_aux:w \iow_now:Nx
5807 }
5808 \cs_new_protected_nopar:Npn \iow_now_buffer_safe:Nx {
5809   \iow_now_buffer_safe_aux:w \iow_now:Nn
5810 }
5811 \cs_new_protected_nopar:Npn \iow_now_buffer_safe_aux:w #1#2#3 {
5812   \group_begin: \tex_newlinechar:D'\ #1#2 {#3} \group_end:
5813 }

```

(End definition for `\iow_now_buffer_safe:Nn`. This function is documented on page 120.)

110.4 Deferred writing

`\iow_shipout_x:Nn` First the easy part, this is the primitive.

`\iow_shipout_x:Nx`

```

5814 \cs_set_eq:NN \iow_shipout_x:Nn \tex_write:D
5815 \cs_generate_variant:Nn \iow_shipout_x:Nn {Nx }

```

(End definition for `\iow_shipout_x:Nn`. This function is documented on page 121.)

`\iow_shipout:Nn` With ϵ -TeX available deferred writing is easy.
`\iow_shipout:Nx`

```
5816 \cs_new_protected_nopar:Npn \iow_shipout:Nn #1#2 {  
5817   \iow_shipout_x:Nn #1 { \exp_not:n {#2} }  
5818 }  
5819 \cs_generate_variant:Nn \iow_shipout:Nn { Nx }
```

(End definition for `\iow_shipout:Nn`. This function is documented on page 121.)

111 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written to an output stream.

```
5820 \cs_new_nopar:Npn \iow_newline: { ^^J }
```

(End definition for `\iow_newline:`. This function is documented on page 121.)

`\iow_char:N` Function to write any escaped char to an output stream.

```
5821 \cs_new:Npn \iow_char:N #1 { \cs_to_str:N #1 }
```

(End definition for `\iow_char:N`. This function is documented on page 121.)

111.1 Reading input

`\if_eof:w` A simple primitive renaming.

```
5822 \cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(End definition for `\if_eof:w`. This function is documented on page 122.)

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.
`\ior_if_eof:NTF` As the pool model means that closed streams are undefined control sequences, the test has two parts.

```
5823 \prg_new_conditional:Nnn \ior_if_eof:N { p , TF , T , F } {  
5824   \cs_if_exist:NTF #1  
5825     { \tex_ifeof:D #1 \prg_return_true: \else: \prg_return_false: \fi: }  
5826     { \prg_return_true: }  
5827 }
```

(End definition for `\ior_if_eof_p:N`. This function is documented on page 122.)

`\ior_to:NN` And here we read from files.

`\ior_gto:NN`

```
5828 \cs_new_protected_nopar:Npn \ior_to:NN #1#2 {  
5829   \tex_read:D #1 to #2  
5830 }  
5831 \cs_new_protected_nopar:Npn \ior_gto:NN {  
5832   \pref_global:D \ior_to:NN  
5833 }  
5834 </initex | package>
```

(End definition for `\ior_to:NN`. This function is documented on page 122.)

112 l3msg implementation

The usual lead-off.

```
5835 <*package>  
5836 \ProvidesExplPackage  
5837   {\filename}{\filedate}{\fileversion}{\filedescription}  
5838 \package_check_loaded_expl:  
5839 </package>  
5840 <*initex | package>
```

L^AT_EX is handling context, so the T_EX “noise” is turned down.

```
5841 \int_set:Nn \tex_errorcontextlines:D { \c_minus_one }
```

112.1 Variables and constants

`\c_msg_error_tl` Header information.

`\c_msg_warning_tl`

`\c_msg_info_tl`

```
5842 \tl_const:Nn \c_msg_error_tl { error }  
5843 \tl_const:Nn \c_msg_warning_tl { warning }  
5844 \tl_const:Nn \c_msg_info_tl { info }
```

(End definition for `\c_msg_error_tl`. This function is documented on page 130.)

`\msg_fatal_text:n` Contextual header/footer information.

`\msg_see_documentation_text:n`

```
5845 \cs_new:Npn \msg_fatal_text:n #1 { Fatal~#1~error }  
5846 \cs_new:Npn \msg_see_documentation_text:n #1  
5847   { See~the~#1~documentation~for~further~information }
```

(End definition for `\msg_fatal_text:n` and `\msg_see_documentation_text:n`. These functions are documented on page ??.)

`\c_msg_coding_error_text_tl` Simple pieces of text for messages.

```

\c_msg_fatal_text_tl
\c_msg_help_text_tl
\c_msg_kernel_bug_text_tl
\c_msg_kernel_bug_more_text_tl
\c_msg_no_info_text_tl
\c_msg_return_text_tl

5848 \tl_const:Nn \c_msg_coding_error_text_tl {
5849   This~is~a~coding~error.
5850   \msg_two_newlines:
5851 }
5852 \tl_const:Nn \c_msg_fatal_text_tl {
5853   This~is~a~fatal~error:~LaTeX~will~abort
5854 }
5855 \tl_const:Nn \c_msg_help_text_tl {
5856   For~immediate~help~type~H~<return>
5857 }
5858 \tl_const:Nn \c_msg_kernel_bug_text_tl {
5859   This~is~a~LaTeX~bug:~check~coding!
5860 }
5861 \tl_const:Nn \c_msg_kernel_bug_more_text_tl {
5862   There~is~a~coding~bug~somewhere~around~here.~\
5863   This~probably~needs~examining~by~an~expert.
5864   \c_msg_return_text_tl
5865 }
5866 \tl_const:Nn \c_msg_no_info_text_tl {
5867   LaTeX~does~not~know~anything~more~about~this~error,~sorry.
5868   \c_msg_return_text_tl
5869 }
5870 \tl_const:Nn \c_msg_return_text_tl {
5871   \
5872   Try~typing~<return>~to~proceed.
5873   \
5874   If~that~doesn't~work,~type~X~<return>~to~quit
5875 }

```

(End definition for `\c_msg_coding_error_text_tl`. This function is documented on page 130.)

`\c_msg_hide_tl<spaces>` An empty variable with a number of (category code 11) periods at the end of its name. This is used to push the T_EX part of an error message “off the screen”.

```

5876 \group_begin:
5877 \char_make_letter:N \.
5878 \tl_to_lowercase:n {
5879   \group_end:
5880   \tl_const:Nn \c_msg_hide_tl.....
5881   {}
5882 }

```

(End definition for `\c_msg_hide_tl<spaces>`.)

`\c_msg_on_line_tl` Text for “on line”.

```

5883 \tl_const:Nn \c_msg_on_line_tl { on~line }

```

(End definition for `\c_msg_on_line_tl`. This function is documented on page 130.)

`\c_msg_text_prefix_tl` Prefixes for storage areas.
`\c_msg_more_text_prefix_tl`

```
5884 \tl_const:Nn \c_msg_text_prefix_tl { msg_text ~>~ }
5885 \tl_const:Nn \c_msg_more_text_prefix_tl { msg_text_more ~>~ }
```

(End definition for `\c_msg_text_prefix_tl`. This function is documented on page 130.)

`\l_msg_class_tl` For holding the current message method and that for redirection.
`\l_msg_current_class_tl`
`\l_msg_current_module_tl`

```
5886 \tl_new:N \l_msg_class_tl
5887 \tl_new:N \l_msg_current_class_tl
5888 \tl_new:N \l_msg_current_module_tl
```

(End definition for `\l_msg_class_tl`. This function is documented on page 130.)

`\l_msg_names_clist` Lists used for filtering.

```
5889 \clist_new:N \l_msg_names_clist
```

(End definition for `\l_msg_names_clist`. This function is documented on page 130.)

`\l_msg_redirect_classes_prop` For filtering messages, a list of all messages and of those which have to be modified is
`\l_msg_redirect_names_prop` required.

```
5890 \prop_new:N \l_msg_redirect_classes_prop
5891 \prop_new:N \l_msg_redirect_names_prop
```

(End definition for `\l_msg_redirect_classes_prop`. This function is documented on page 130.)

`\l_msg_redirect_classes_clist` To prevent an infinite loop.

```
5892 \clist_new:N \l_msg_redirect_classes_clist
```

(End definition for `\l_msg_redirect_classes_clist`. This function is documented on page 130.)

`\l_msg_tmp_tl` A scratch variable.

```
5893 \tl_new:N \l_msg_tmp_tl
```

(End definition for `\l_msg_tmp_tl`. This function is documented on page ??.)

112.2 Output helper functions

`\msg_line_number:` For writing the line number nicely.
`\msg_line_context:`

```
5894 \cs_new_nopar:Npn \msg_line_number: {
5895   \toks_use:N \tex_inputlineno:D
5896 }
5897 \cs_new_nopar:Npn \msg_line_context: {
5898   \c_msg_on_line_tl
5899   \c_space_tl
5900   \msg_line_number:
5901 }
```


(End definition for `\msg_line_number:`. This function is documented on page 127.)

`\msg_newline:` Always forces a new line.
`\msg_two_newlines:`

```
5902 \cs_new_nopar:Npn \msg_newline: { ^^J }
5903 \cs_new_nopar:Npn \msg_two_newlines: { ^^J ^^J }
```

(End definition for `\msg_newline:`. This function is documented on page 127.)

112.3 Generic functions

The lowest level functions make no assumptions about modules, *etc.*

`\msg_generic_new:nnn` Creating a new message is basically the same as the non-checking version, and so after a
`\msg_generic_new:nn` check everything hands over.

```
5904 \cs_new_protected_nopar:Npn \msg_generic_new:nnn #1 {
5905   \chk_if_free_cs:c { \c_msg_text_prefix_tl #1 :xxxx }
5906   \msg_generic_set:nnn {#1}
5907 }
5908 \cs_new_protected_nopar:Npn \msg_generic_new:nn #1 {
5909   \chk_if_free_cs:c { \c_msg_text_prefix_tl #1 :xxxx }
5910   \msg_generic_set:nn {#1}
5911 }
```

(End definition for `\msg_generic_new:nnn`. This function is documented on page 127.)

`\msg_generic_set:nnn` Creating a message is quite simple. There must be a short text part, while the longer
`\msg_generic_set:nn` text may or may not be available.
`\msg_generic_set_clist:n`

```
5912 \cs_new_protected_nopar:Npn \msg_generic_set:nnn #1#2#3 {
5913   \msg_generic_set_clist:n {#1}
5914   \cs_set:cpn { \c_msg_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#2}
5915   \cs_set:cpn { \c_msg_more_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#3}
5916 }
5917 \cs_new_protected_nopar:Npn \msg_generic_set:nn #1#2 {
5918   \msg_generic_set_clist:n {#1}
5919   \cs_set:cpn { \c_msg_text_prefix_tl #1 :xxxx } ##1##2##3##4 {#2}
5920   \cs_set_eq:cN { \c_msg_more_text_prefix_tl #1 :xxxx } \c_undefined
5921 }
5922 \cs_new_protected_nopar:Npn \msg_generic_set_clist:n #1 {
5923   \clist_if_in:NnF \l_msg_names_clist { // #1 / } {
5924     \clist_put_right:Nn \l_msg_names_clist { // #1 / }
5925   }
5926 }
```

(End definition for `\msg_generic_set:nnn`. This function is documented on page 127.)

`\msg_direct_interrupt:xxxxx`
`\msg_direct_interrupt:n`

The low-level interruption macro is rather opaque, unfortunately. The idea here is to create a message which hides all of TeX's own information by filling the output up with dots. To achieve this, dots have to be letters. The odd `\c_msg_hide_tl<dots>` actually does the hiding: it is the large run of dots in the name that is important here. The meaning of `\` is altered so that the explanation text is a simple run whilst the initial error has line-continuation shown.

```

5927 \group_begin:
5928   \char_set_lccode:nn {'\&} {'\ } % {
5929   \char_set_lccode:w '\} = '\ \scan_stop:
5930   \char_set_lccode:w '\& = '\!\scan_stop:
5931   \char_make_active:N \&
5932   \char_make_letter:N \.
5933   \tl_to_lowercase:n{
5934     \group_end:
5935     \cs_new_protected:Npn \msg_direct_interrupt:xxxxx #1#2#3#4#5 {
5936       \group_begin:
5937         \cs_set_nopar:Npn \ \ { \msg_newline: }
5938         \cs_set_eq:NN \ \c_space_tl
5939         \tl_set:Nx \l_tmpa_tl {#5}
5940         \tl_set:Nx \l_tmpb_tl {\c_msg_no_info_text_tl}
5941         \msg_direct_interrupt_aux:n {#5}
5942         \tex_errhelp:D \l_msg_tmp_tl
5943         \cs_set_nopar:Npn \ \ { \msg_newline: !~#3}
5944         \iow_term:x
5945           {\msg_newline: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!}
5946         \cs_set:Npn & {
5947           \tex_errmessage:D{
5948             \ \ #1 \ \ \ \ #2 \ \ \ \ #4
5949             \tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
5950               { . \ \ \c_msg_help_text_tl }
5951             \c_msg_hide_tl.....
5952           }
5953         }
5954         &
5955       \group_end:
5956     }
5957   }

5958 \cs_new_protected:Npn \msg_direct_interrupt_aux:n #1 {
5959   \cs_set_nopar:Npn \ \ { \msg_newline: |~}
5960   \tl_if_empty:nTF {#1} {
5961     \tl_set:Nx \l_msg_tmp_tl { { \c_msg_no_info_text_tl } }
5962   }{
5963     \tl_set:Nx \l_msg_tmp_tl { {
5964       |,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\
5965       #1^^J
5966       |.....^^J
5967     } }
5968   }

```

```
5969 }
```

(End definition for `\msg_direct_interrupt:xxxx`. This function is documented on page 128.)

```
\msg_direct_log:xx
\msg_direct_term:xx
```

Printing to the log or terminal without a stop is rather easier.

```
5970 \cs_new_protected:Npn \msg_direct_log:xx #1#2 {
5971   \group_begin:
5972     \cs_set:Npn \ { \msg_newline: #2 }
5973     \cs_set_eq:NN \ \c_space_tl
5974     \iow_log:x { #1 \msg_newline: }
5975   \group_end:
5976 }
5977 \cs_new_protected:Npn \msg_direct_term:xx #1#2 {
5978   \group_begin:
5979     \cs_set:Npn \ { \msg_newline: #2 }
5980     \cs_set_eq:NN \ \c_space_tl
5981     \iow_term:x { #1 \msg_newline: }
5982   \group_end:
5983 }
```

(End definition for `\msg_direct_log:xx`. This function is documented on page 128.)

112.4 General functions

The main functions for messaging are built around the separation of module from the message name. These have short names as they will be widely used.

```
\msg_new:nnnn
\msg_new:nnn
\msg_set:nnnn
\msg_set:nnn
```

For making messages: all aliases.

```
5984 \cs_new_protected_nopar:Npn \msg_new:nnnn #1#2 {
5985   \msg_generic_new:nnn { #1 / #2 }
5986 }
5987 \cs_new_protected_nopar:Npn \msg_new:nnn #1#2 {
5988   \msg_generic_new:nn { #1 / #2 }
5989 }
5990 \cs_new_protected_nopar:Npn \msg_set:nnnn #1#2 {
5991   \msg_generic_set:nnn { #1 / #2 }
5992 }
5993 \cs_new_protected_nopar:Npn \msg_set:nnn #1#2 {
5994   \msg_generic_set:nn { #1 / #2 }
5995 }
```

(End definition for `\msg_new:nnnn`. This function is documented on page 124.)

```
\msg_class_new:nn
\msg_class_set:nn
```

Creating a new class produces three new functions, with varying numbers of arguments. The `\msg_class_loop:n` function is set up so that redirection will work as desired.

```
5996 \cs_new_protected_nopar:Npn \msg_class_new:nn #1 {
```

```

5997 \chk_if_free_cs:c { msg_ #1 :nnxxxx }
5998 \prop_new:c { l_msg_redirect_ #1 _prop }
5999 \msg_class_set:nn {#1}
6000 }
6001 \cs_new_protected_nopar:Npn \msg_class_set:nn #1#2 {
6002 \prop_clear:c { l_msg_redirect_ #1 _prop }
6003 \cs_set_protected:cpn { msg_ #1 :nnxxxx } ##1##2##3##4##5##6 {
6004 \msg_use:nnnnxxxx {#1} {#2} {##1} {##2} {##3} {##4} {##5} {##6}
6005 }
6006 \cs_set_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5 {
6007 \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { }
6008 }
6009 \cs_set_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4 {
6010 \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { }
6011 }
6012 \cs_set_protected:cpx { msg_ #1 :nnx } ##1##2##3 {
6013 \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { }
6014 }
6015 \cs_set_protected:cpx { msg_ #1 :nn } ##1##2 {
6016 \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} { } { } { } { }
6017 }
6018 }

```

(End definition for `\msg_class_new:nn`. This function is documented on page 124.)

`\msg_use:nnnnxxxx` The main message-using macro creates two auxiliary functions: one containing the code for the message, and the second a loop function. There is then a hand-off to the system for checking if redirection is needed.

```

6019 \cs_new_protected:Npn \msg_use:nnnnxxxx #1#2#3#4#5#6#7#8 {
6020 \cs_set_nopar:Npn \msg_use_code: {
6021 \clist_clear:N \l_msg_redirect_classes_clist
6022 #2
6023 }
6024 \cs_set:Npn \msg_use_loop:n ##1 {
6025 \clist_if_in:NnTF \l_msg_redirect_classes_clist {#1} {
6026 \msg_kernel_error:nn { msg } { redirect-loop } {#1}
6027 }{
6028 \clist_put_right:Nn \l_msg_redirect_classes_clist {#1}
6029 \cs_if_exist:cTF { msg_ ##1 :nnxxxx } {
6030 \use:c { msg_ ##1 :nnxxxx } {#3} {#4} {#5} {#6} {#7} {#8}
6031 }{
6032 \msg_kernel_error:nnx { msg } { message-class-unknown } {##1}
6033 }
6034 }
6035 }
6036 \cs_if_exist:cTF { \c_msg_text_prefix_tl #3 / #4 :xxxx } {
6037 \msg_use_aux:nnn {#1} {#3} {#4}
6038 }{
6039 \msg_kernel_error:nnxx { msg } { message-unknown } {#3} {#4}

```

```

6040 }
6041 }

```

(End definition for \msg_use:nnnnxxx.)

\msg_use_code: Blank definitions are initially created for these functions.

```

\msg_use_loop:
6042 \cs_new_nopar:Npn \msg_use_code: { }
6043 \cs_new:Npn \msg_use_loop:n #1 { }

```

(End definition for \msg_use_code:.)

\msg_use_aux:nnn The first auxiliary macro looks for a match by name: the most restrictive check.

```

6043 \cs_new_protected_nopar:Npn \msg_use_aux:nnn #1#2#3 {
6044   \tl_set:Nn \l_msg_current_class_tl {#1}
6045   \tl_set:Nn \l_msg_current_module_tl {#2}
6046   \prop_if_in:NnTF \l_msg_redirect_names_prop { // #2 / #3 / } {
6047     \msg_use_loop_check:nn { names } { // #2 / #3 / }
6048   }{
6049     \msg_use_aux:nn {#1} {#2}
6050   }
6051 }
6052 }

```

(End definition for \msg_use_aux:nnn.)

\msg_use_aux:nn The second function checks for general matches by module or for all modules.

```

6053 \cs_new_protected_nopar:Npn \msg_use_aux:nn #1#2 {
6054   \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } {#2} {
6055     \msg_use_loop_check:nn {#1} {#2}
6056   }{
6057     \prop_if_in:cnTF { l_msg_redirect_ #1 _prop } { * } {
6058       \msg_use_loop_check:nn {#1} { * }
6059     }{
6060       \msg_use_code:
6061     }
6062   }
6063 }

```

(End definition for \msg_use_aux:nn.)

\msg_use_loop_check:nn When checking whether to loop, the same code is needed in a few places.

```

6064 \cs_new_protected:Npn \msg_use_loop_check:nn #1#2 {
6065   \prop_get:cnN { l_msg_redirect_ #1 _prop } {#2} \l_msg_class_tl
6066   \tl_if_eq:NNTF \l_msg_current_class_tl \l_msg_class_tl {
6067     \msg_use_code:
6068   }{
6069     \msg_use_loop:n { \l_msg_class_tl }
6070   }
6071 }

```

(End definition for `\msg_use_loop_check:nn`.)

`\msg_fatal:nnxxxx`
`\msg_fatal:nnxxx`
`\msg_fatal:nnxx`
`\msg_fatal:nnx`
`\msg_fatal:nn`

For fatal errors, after the error message T_EX bails out.

```

6072 \msg_class_new:nn { fatal } {
6073   \msg_direct_interrupt:xxxxx
6074   { \msg_fatal_text:n {#1} : ~ "#2" }
6075   {
6076     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
6077   }
6078   {}
6079   { \msg_see_documentation_text:n {#1} }
6080   { \c_msg_fatal_text_tl }
6081   \tex_end:D
6082 }

```

(End definition for `\msg_fatal:nnxxxx`. This function is documented on page 124.)

`\msg_error:nnxxxx`
`\msg_error:nnxxx`
`\msg_error:nnxx`
`\msg_error:nnx`
`\msg_error:nn`

For an error, the interrupt routine is called, then any recovery code is tried.

```

6083 \msg_class_new:nn { error } {
6084   \msg_direct_interrupt:xxxxx
6085   { #1~ \c_msg_error_tl : ~ "#2" }
6086   {
6087     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
6088   }
6089   {}
6090   { \msg_see_documentation_text:n {#1} }
6091   {
6092     \cs_if_exist:cTF { \c_msg_more_text_prefix_tl #1 / #2 :xxxx }
6093     {
6094       \use:c { \c_msg_more_text_prefix_tl #1 / #2 :xxxx }
6095       {#3} {#4} {#5} {#6}
6096     }
6097     { \c_msg_no_info_text_tl }
6098   }
6099 }

```

(End definition for `\msg_error:nnxxxx`. This function is documented on page 125.)

`\msg_warning:nnxxxx`
`\msg_warning:nnxxx`
`\msg_warning:nnxx`
`\msg_warning:nnx`
`\msg_warning:nn`

Warnings are printed to the terminal.

```

6100 \msg_class_new:nn { warning } {
6101   \msg_direct_term:xx {
6102     \c_space_tl #1 ~ \c_msg_warning_tl :~
6103     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
6104   }
6105   { ( #1 ) \c_space_tl \c_space_tl }
6106 }

```

(End definition for `\msg_warning:nnxxxx`. This function is documented on page 125.)

```

\msg_info:nnxxxx Information only goes into the log.
\msg_info:nnxxx
\msg_info:nnxx
\msg_info:nnx
\msg_info:nn
6107 \msg_class_new:nn { info } {
6108   \msg_direct_log:xx {
6109     \c_space_tl #1~\c_msg_info_tl :~
6110     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
6111   }
6112   { ( #1 ) \c_space_tl \c_space_tl }
6113 }

```

(End definition for `\msg_info:nnxxxx`. This function is documented on page 125.)

```

\msg_log:nnxxxx “Log” data is very similar to information, but with no extras added.
\msg_log:nnxxx
\msg_log:nnxx
\msg_log:nnx
\msg_log:nn
6114 \msg_class_new:nn { log } {
6115   \msg_direct_log:xx {
6116     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
6117   }
6118   { }
6119 }

```

(End definition for `\msg_log:nnxxxx`. This function is documented on page 125.)

```

\msg_trace:nnxxxx Trace data is the same as log data, more or less
\msg_trace:nnxxx
\msg_trace:nnxx
\msg_trace:nnx
\msg_trace:nn
6120 \msg_class_new:nn { trace } {
6121   \msg_direct_log:xx {
6122     \use:c { \c_msg_text_prefix_tl #1 / #2 :xxxx } {#3} {#4} {#5} {#6}
6123   }
6124   { }
6125 }

```

(End definition for `\msg_trace:nnxxxx`. This function is documented on page 126.)

```

\msg_none:nnxxxx The none message type is needed so that input can be gobbled.
\msg_none:nnxxx
\msg_none:nnxx
\msg_none:nnx
\msg_none:nn
6126 \msg_class_new:nn { none } { }

```

(End definition for `\msg_none:nnxxxx`. This function is documented on page 126.)

112.5 Redirection functions

`\msg_redirect_class:nn` Converts class one into class two.

```

6127 \cs_new_protected_nopar:Npn \msg_redirect_class:nn #1#2 {
6128   \prop_put:cnn { l_msg_redirect_ #1 _prop } { * } {#2}
6129 }

```

(End definition for `\msg_redirect_class:nn`. This function is documented on page 126.)

`\msg_redirect_module:nnn` For when all messages of a class should be altered for a given module.

```

6130 \cs_new_protected_nopar:Npn \msg_redirect_module:nnn #1#2#3 {
6131   \prop_put:cnn { l_msg_redirect_ #2 _prop } {#1} {#3}
6132 }

```

(End definition for `\msg_redirect_module:nnn`. This function is documented on page 126.)

`\msg_redirect_name:nnn` Named message will always use the given class.

```

6133 \cs_new_protected_nopar:Npn \msg_redirect_name:nnn #1#2#3 {
6134   \prop_put:Nnn \l_msg_redirect_names_prop { // #1 / #2 / } {#3}
6135 }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 126.)

112.6 Kernel-specific functions

`\msg_kernel_new:nnnn` The kernel needs some messages of its own. These are created using pre-built functions.
`\msg_kernel_new:nnn` Two functions are provided: one more general and one which only has the short text
`\msg_kernel_set:nnnn` part.
`\msg_kernel_set:nnn`

```

6136 \cs_new_protected_nopar:Npn \msg_kernel_new:nnnn #1#2 {
6137   \msg_new:nnnn { LaTeX } { #1 / #2 }
6138 }
6139 \cs_new_protected_nopar:Npn \msg_kernel_new:nnn #1#2 {
6140   \msg_new:nnn { LaTeX } { #1 / #2 }
6141 }
6142 \cs_new_protected_nopar:Npn \msg_kernel_set:nnnn #1#2 {
6143   \msg_set:nnnn { LaTeX } { #1 / #2 }
6144 }
6145 \cs_new_protected_nopar:Npn \msg_kernel_set:nnn #1#2 {
6146   \msg_set:nnn { LaTeX } { #1 / #2 }
6147 }

```

(End definition for `\msg_kernel_new:nnnn`. This function is documented on page 128.)

`\msg_kernel_classes_new:n` Quickly make the fewer-arguments versions.

```

6148 \cs_new_protected_nopar:Npn \msg_kernel_classes_new:n #1 {
6149   \cs_new_protected:cpx { msg_kernel_ #1 :nnxxx } ##1##2##3##4##5
6150   {
6151     \exp_not:c { msg_kernel_ #1 :nnxxxx }
6152     {##1} {##2} {##3} {##4} {##5} { }
6153   }
6154   \cs_new_protected:cpx { msg_kernel_ #1 :nnxx } ##1##2##3##4
6155   {
6156     \exp_not:c { msg_kernel_ #1 :nnxxxx }
6157     {##1} {##2} {##3} {##4} { } { }

```



```

6158 }
6159 \cs_new_protected:cpx { msg_kernel_ #1 :nnx } ##1##2##3
6160 {
6161   \exp_not:c { msg_kernel_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { }
6162 }
6163 \cs_new_protected:cpx { msg_kernel_ #1 :nn } ##1##2
6164 {
6165   \exp_not:c { msg_kernel_ #1 :nnxxxx } {##1} {##2} { } { } { } { }
6166 }
6167 }

```

(End definition for `\msg_kernel_classes_new:n`.)

`\msg_kernel_fatal:nnxxxx` Fatal kernel errors cannot be re-defined.

```

\msg_kernel_fatal:nnxxxx
\msg_kernel_fatal:nnxxx
\msg_kernel_fatal:nnxx
\msg_kernel_fatal:nnx
\msg_kernel_fatal:nn
6168 \cs_new_protected:Npn \msg_kernel_fatal:nnxxxx #1#2#3#4#5#6 {
6169   \msg_direct_interrupt:xxxxx
6170   { \msg_fatal_text:n {LaTeX} }
6171   {
6172     \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 :xxxx }
6173     {#3} {#4} {#5} {#6}
6174   }
6175   {}
6176   { \msg_see_documentation_text:n {LaTeX3} }
6177   { \c_msg_fatal_text_tl }
6178   \tex_end:D
6179 }
6180 \msg_kernel_classes_new:n { fatal }

```

(End definition for `\msg_kernel_fatal:nnxxxx`. This function is documented on page 128.)

`\msg_kernel_error:nnxxxx` Neither can kernel errors.

```

\msg_kernel_error:nnxxxx
\msg_kernel_error:nnxxx
\msg_kernel_error:nnxx
\msg_kernel_error:nnx
\msg_kernel_error:nn
6181 \cs_new_protected:Npn \msg_kernel_error:nnxxxx #1#2#3#4#5#6 {
6182   \msg_direct_interrupt:xxxxx
6183   { LaTeX~\c_msg_error_tl \c_space_tl "#2" }
6184   {
6185     \use:c { \c_msg_text_prefix_tl LaTeX / #1 / #2 :xxxx }
6186     {#3} {#4} {#5} {#6}
6187   }
6188   {}
6189   { \msg_see_documentation_text:n {LaTeX3} }
6190   {
6191     \cs_if_exist:cTF
6192     { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 :xxxx }
6193     {
6194       \use:c { \c_msg_more_text_prefix_tl LaTeX / #1 / #2 :xxxx }
6195       {#3} {#4} {#5} {#6}
6196     }
6197     { \c_msg_no_info_text_tl }

```

```

6198     }
6199   }
6200   \msg_kernel_classes_new:n { error }

```

(End definition for `\msg_kernel_error:nxxxxx`. This function is documented on page 129.)

```

\msg_kernel_warning:nxxxxx
\msg_kernel_warning:nxxxx
\msg_kernel_warning:nxxx
\msg_kernel_warning:nxx
\msg_kernel_warning:nnx
\msg_kernel_warning:nn
\msg_kernel_info:nxxxxx
\msg_kernel_info:nxxxx
\msg_kernel_info:nxxx
\msg_kernel_info:nxx
\msg_kernel_info:nnx
\msg_kernel_info:nn

```

Life is much more simple for warnings and information messages, as these are just short-cuts to the standard classes.

```

6201   \cs_new_protected_nopar:Npn \msg_kernel_warning:nxxxxx #1#2 {
6202     \msg_warning:nxxxxx { LaTeX } { #1 / #2 }
6203   }
6204   \msg_kernel_classes_new:n { warning }
6205   \cs_new_protected_nopar:Npn \msg_kernel_info:nxxxxx #1#2 {
6206     \msg_info:nxxxxx { LaTeX } { #1 / #2 }
6207   }
6208   \msg_kernel_classes_new:n { info }

```

(End definition for `\msg_kernel_warning:nxxxxx`. This function is documented on page 129.)

Error messages needed to actually implement the message system itself.

```

6209   \msg_kernel_new:nnnn { msg } { message-unknown }
6210   { Unknown~message~'#2'~for~module~'#1'. }
6211   {
6212     \c_msg_coding_error_text_tl
6213     LaTeX~was~asked~to~display~a~message~called~'#2'\
6214     by~the~module~'#1'~module:~this~message~does~not~exist.
6215     \c_msg_return_text_tl
6216   }
6217   \msg_kernel_new:nnnn { msg } { message-class-unknown }
6218   { Unknown~message~class~'#1'. }
6219   {
6220     LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\
6221     this~was~never~defined.
6222   }
6223   \c_msg_return_text_tl
6224   }
6225   \msg_kernel_new:nnnn { msg } { redirect-loop }
6226   { Message~redirection~loop~for~message~class~'#1'. }
6227   {
6228     LaTeX~has~been~asked~to~redirect~messages~in~an~infinite~loop.\
6229     The~original~message~here~has~been~lost.
6230     \c_msg_return_text_tl
6231   }

```

```

\msg_kernel_bug:x

```

The L^AT_EX coding bug error gets re-visited here.

```

6232   \cs_set_protected:Npn \msg_kernel_bug:x #1 {
6233     \msg_direct_interrupt:xxxxx
6234     { \c_msg_kernel_bug_text_tl }

```

```

6235 { #1 }
6236 {}
6237 { \msg_see_documentation_text:n {LaTeX3} }
6238 { \c_msg_kernel_bug_more_text_tl }
6239 }

```

(End definition for `\msg_kernel_bug:x`. This function is documented on page 129.)

```

6240 </initex | package>

```

113 l3xref implementation

113.1 Internal functions and variables

<code>\g_xref_all_curr_immediate_fields_prop</code> <code>\g_xref_all_curr_deferred_fields_prop</code>	What they say they are :)
---	---------------------------

<code>\xref_write</code>	A stream for writing cross references, although they are not required to be in a separate file.
--------------------------	---

<code>\xref_define_label:nn</code>	<code>\xref_define_label:nn {<name>} {<plist contents>}</code>
------------------------------------	--

Define the property list for each label; used internally by `\xref_set_label:n`.

113.2 Module code

We start by ensuring that the required packages are loaded.

```

6241 <*package>
6242 \ProvidesExplPackage
6243   {\filename}{\filedate}{\fileversion}{\filedescription}
6244 \package_check_loaded_expl:
6245 </package>
6246 <*initex | package>

```

There are two kinds of information, namely information which is *immediate* like a section title and then there's *deferred* information like page numbers. Each reference type belong to one of these categories, which we save internally as the property lists `\g_xref_all_curr_immediate_fields_prop` and `\g_xref_all_curr_deferred_fields_prop` and the reference type `<xyz>` exists as the key-info pair `\xref_<xyz>_key {\l_xref_curr_<xyz>_tl}` on one of these lists. This way each new entry type is just added as another key-info pair.

When the cross references are generated at the beginning of the document each will turn into a control sequence. Thus `\label{mylab}` will internally refer to the property list `\g_xref_mylab_prop`.

The extraction of values from this property list can be done in several different ways but we want to keep the operation expandable. Therefore we use a dedicated function for each type of cross reference, which looks like this:

```
\xref_get_value_xyz_aux:w -> #1 \xref_xyz_key #2#3\q_nil{#2}
```

This will throw away all the bits we don't need. In case `xyz` is the first on the `mylab` property list `#1` is empty, if it's the last key-info pair `#3` is empty. The value of the field can be extracted with the function `\xref_get_value:nn` where the first argument is the type and the second the label name so here it would be `\xref_get_value:nn {xyz}{mylab}`.

`\g_xref_all_curr_immediate_fields_prop`
`\g_xref_all_curr_deferred_fields_prop`

The two main property lists for storing information. They contain key-info pairs for all known types.

```
6247 \prop_new:N \g_xref_all_curr_immediate_fields_prop
6248 \prop_new:N \g_xref_all_curr_deferred_fields_prop
```

(End definition for `\g_xref_all_curr_immediate_fields_prop`. This function is documented on page 380.)

`\xref_new:nn`
`\xref_deferred_new:nn`
`\xref_new_aux:nnn`

Setting up a new cross reference type is fairly straight forward when we follow the game plan mentioned earlier.

```
6249 \cs_new_nopar:Npn \xref_new:nn {\xref_new_aux:nnn{immediate}}
6250 \cs_new_nopar:Npn \xref_deferred_new:nn {\xref_new_aux:nnn{deferred}}
6251 \cs_new_nopar:Npn \xref_new_aux:nnn #1#2#3{
```

First put the new type in the relevant property list.

```
6252 \prop_gput:ccx {g_xref_all_curr_ #1 _fields_prop}
6253 { xref_ #2 _key }
6254 { \exp_not:c {l_xref_curr_#2_t1 }}}
```

Then define the key to be a protected macro.¹¹

```
6255 \cs_set_protected_nopar:cpn { xref_#2_key }{ }
6256 \tl_new:cn{l_xref_curr_#2_t1}{#3}
```

Now for the function extracting the value of a reference. We could do this with a simple `\prop_if_in` thing put since we want to do things in an expandable way we make a separate grabber for each type—this is also faster. The grabber function can be defined

¹¹We could also set it equal to `\scan_stop:` but this just feels “cleaner”.

by using an intricate construction of `\exp_after:wN` and other goodies but I prefer readable code. The end result for the input `xyz` is

```
\cs_set_nopar:Npn\xref_get_value_xyz_aux:w #1\xref_xyz_key #2#3\q_nil{#2}
```

```
6257 \toks_set:Nx \l_tmpa_toks {
6258   \exp_not:n { \cs_set_nopar:cpn {xref_get_value_#2_aux:w} ##1 }
6259   \exp_not:N \q_prop
6260   \exp_not:c { xref_#2_key }
6261   \exp_not:N \q_prop
6262 }
6263 \toks_use:N \l_tmpa_toks ##2 ##3\q_nil {##2}
6264 }
```

(End definition for `\xref_new:nn`. This function is documented on page 131.)

`\xref_get_value:nn` Getting the correct value for a given label-type pair is a matter of connecting the correct grabber functions and property list.

```
6265 \cs_new_nopar:Npn \xref_get_value:nn #1#2 {
6266   \cs_if_exist:cTF{g_xref_#2_prop}
6267   {
```

This next expansion may look a little weird but it isn't if you think about it!

```
6268   \exp_args:NcNc \exp_after:wN {xref_get_value_#1_aux:w}
6269   \toks_use:N {g_xref_#2_prop}
```

Better put in the stop marker.

```
6270   \q_nil
6271 }
6272 {??}
6273 }
```

Temporary! We expand the property list and so we can't have the `\q_prop` marker just expand!

```
6274 \cs_set_nopar:Npn \exp_after:cc #1#2 {
6275   \exp_after:wN \exp_after:wN
6276   \cs:w #1\exp_after:wN\cs_end: \cs:w #2\cs_end:
6277 }
6278 \cs_set_protected:Npn \q_prop {\q_prop}
```

(End definition for `\xref_get_value:nn`. This function is documented on page 131.)

`\xref_define_label:nn` Define the property list for each label. We better do this in two steps because the special catcode regime is in effect and since some of the info fields are very likely to contain actual text, we better make sure spaces aren't ignored! As for the meaning of other characters

then it is a possibility to also have a field containing catcode instructions which can then be activated with `\etex_scantokens:D`.

```

6279 \cs_new_protected_nopar:Npn \xref_define_label:nn {
6280   \group_begin:
6281     \char_set_catcode:nn {'\ } \c_ten
6282     \xref_define_label_aux:nn
6283 }

```

If the label is already taken we have a multiply defined label and we should do something about it. For now we don't do anything spectacular.

```

6284 \cs_new_nopar:Npn \xref_define_label_aux:nn #1#2 {
6285   \cs_if_free:cTF{g_xref_#1_prop}
6286   {\prop_new:c{g_xref_#1_prop}}{\WARNING}
6287   \toks_gset:cn{g_xref_#1_prop}{#2}
6288   \group_end:
6289 }

```

(End definition for `\xref_define_label:nn`. This function is documented on page 380.)

`\xref_set_label:n` Then the generic command for setting a label. We expand the immediate labels fully before calling the write function but make sure the deferred fields aren't expanded just yet. Due to property lists being implemented as token list registers we must expand the 'immediate' fields twice.

```

6290 \cs_set_nopar:Npn \xref_set_label:n #1{
6291   \cs_set_nopar:Npx \xref_tmp:w{\toks_use:N\g_xref_all_curr_immediate_fields_prop}
6292   \exp_args:NNx\iow_shipout_x:Nn \xref_write{
6293     \xref_define_label:nn {#1} {
6294       \xref_tmp:w
6295       \toks_use:N\g_xref_all_curr_deferred_fields_prop
6296     }
6297   }
6298 }

```

(End definition for `\xref_set_label:n`. This function is documented on page 131.)

That's it (for now).

```

6299 </initex | package>

```

114 l3xref test file

```

6300 <*testfile>
6301 \documentclass{article}
6302 \usepackage{l3xref}
6303 \ExplSyntaxOn
6304 \cs_set_nopar:Npn \startrecording {\iow_open:Nn \xref_write {\jobname.xref}}
6305 \cs_set_nopar:Npn \DefineCrossReferences {

```

```

6306 \group_begin:
6307     \ExplSyntaxNamesOn
6308     \InputIfFileExists{\jobname.xref}{\}{}
6309 \group_end:
6310 }
6311 \AtBeginDocument{\DefineCrossReferences\startrecording}
6312
6313 \xref_new:nn {name}{}
6314 \cs_set_nopar:Npn \setname{\tl_set:Nn\l_xref_curr_name_tl}
6315 \cs_set_nopar:Npn \getname{\xref_get_value:nn{name}}
6316
6317 \xref_deferred_new:nn {page}{\thepage}
6318 \cs_set_nopar:Npn \getpage{\xref_get_value:nn{page}}
6319
6320 \xref_deferred_new:nn {valuepage}{\number\value{page}}
6321 \cs_set_nopar:Npn \getvaluepage{\xref_get_value:nn{valuepage}}
6322
6323 \cs_set_eq:NN \setlabel \xref_set_label:n
6324
6325 \ExplSyntaxOff
6326 \begin{document}
6327 \pagenumbering{roman}
6328
6329 Text\setname{This is a name}\setlabel{testlabel1}. More
6330 text\setname{This is another name}\setlabel{testlabel2}. \clearpage
6331
6332 Text\setname{This is a third name}\setlabel{testlabel3}. More
6333 text\setname{Hello World!}\setlabel{testlabel4}. \clearpage
6334
6335 \pagenumbering{arabic}
6336
6337 Text\setname{Name 5}\setlabel{testlabel5}. More text\setname{Name
6338 6}\setlabel{testlabel6}. \clearpage
6339
6340 Text\setname{Name 7}\setlabel{testlabel 7}. More text\setname{Name
6341 8}\setlabel{testlabel8}. \clearpage
6342
6343 Now let's extract some values. \getname{testlabel1} on page
6344 \getpage{testlabel1} with value \getvaluepage{testlabel1}.
6345
6346 Now let's extract some values. \getname{testlabel 7} on page
6347 \getpage{testlabel 7} with value \getvaluepage{testlabel 7}.
6348 \end{document}
6349 \testfile

```

115 l3keyval implementation

<code>\KV_sanitize_outerlevel_active_equals:N</code> <code>\KV_sanitize_outerlevel_active_commas:N</code>	<code>\KV_sanitize_outerlevel_active_equals:N <tl var.></code>
--	--

Replaces catcode other = and , within a $\langle tl\ var.\rangle$ with active characters.

<code>\KV_remove_surrounding_spaces:nw</code> <code>\KV_remove_surrounding_spaces_auxi:w *</code>	<code>\KV_remove_surrounding_spaces:nw <tl> <token list> \q_nil</code> <code>\KV_remove_surrounding_spaces_auxi:w <token list> \Q_3</code>
--	---

Removes a possible leading space plus a possible ending space from a $\langle token\ list\rangle$. The first version (which is not used in the code) stores it in $\langle tl\rangle$.

<code>\KV_add_value_element:w</code> <code>\KV_set_key_element:w</code>	<code>\KV_set_key_element:w <token list> \q_nil</code> <code>\KV_add_value_element:w \q_stop <token list> \q_nil</code>
--	--

Specialised functions to strip spaces from their input and set the token registers $\backslash l_KV_currkey_tl$ or $\backslash l_KV_currval_tl$ respectively.

<code>\KV_split_key_value_current:w</code> <code>\KV_split_key_value_space_removal:w</code> <code>\KV_split_key_value_space_removal_detect_error:wTF</code> <code>\KV_split_key_value_no_space_removal:w</code>	<code>\KV_split_key_value_current:w ...</code>
--	--

These functions split keyval lists into chunks depending which sanitising method is being used. $\backslash KV_split_key_value_current:w$ is $\backslash cs_set_eq:NN$ to whichever is appropriate.

115.1 Module code

We start by ensuring that the required packages are loaded.

```

6350 <*package>
6351 \ProvidesExplPackage
6352   {\filename}{\filedate}{\fileversion}{\filedescription}
6353 \package_check_loaded_expl:
6354 </package>
6355 <*initex | package>

```

$\backslash l_KV_tmpa_tl$ $\backslash l_KV_tmpb_tl$ $\backslash c_KV_single_equal_sign_tl$	<p>Various useful things.</p> <pre> 6356 \tl_new:N \l_KV_tmpa_tl 6357 \tl_new:N \l_KV_tmpb_tl 6358 \tl_const:Nn \c_KV_single_equal_sign_tl { = } </pre>
---	---

(End definition for `\l_KV_tmpa_tl`. This function is documented on page 134.)

`\l_KV_parse_tl` Some more useful things.
`\l_KV_currkey_tl`
`\l_KV_currval_tl`

```
6359 \tl_new:N \l_KV_parse_tl
6360 \tl_new:N \l_KV_currkey_tl
6361 \tl_new:N \l_KV_currval_tl
```

(End definition for `\l_KV_parse_tl`. This function is documented on page 134.)

`\l_KV_level_int` This is used to track how deeply nested calls to the keyval processor are, so that the correct functions are always in use.

```
6362 \int_new:N \l_KV_level_int
```

(End definition for `\l_KV_level_int`. This function is documented on page ??.)

`\l_KV_remove_one_level_of_braces_bool` A boolean to control

```
6363 \bool_new:N \l_KV_remove_one_level_of_braces_bool
6364 \bool_set_true:N \l_KV_remove_one_level_of_braces_bool
```

(End definition for `\l_KV_remove_one_level_of_braces_bool`. This function is documented on page 133.)

`\KV_process_space_removal_sanitiz:NNn`
`\KV_process_space_removal_no_sanitiz:NNn`
`\KV_process_no_space_removal_no_sanitiz:NNn`
`\KV_process_aux:NNNn` The wrapper function takes care of assigning the appropriate `elt` functions before and after the parsing step. In that way there is no danger of a mistake with the wrong functions being used.

```
6365 \cs_new_protected_nopar:Npn \KV_process_space_removal_sanitiz:NNn {
6366   \KV_process_aux:NNNn \KV_parse_space_removal_sanitiz:n
6367 }
6368 \cs_new_protected_nopar:Npn \KV_process_space_removal_no_sanitiz:NNn {
6369   \KV_process_aux:NNNn \KV_parse_space_removal_no_sanitiz:n
6370 }
6371 \cs_new_protected_nopar:Npn \KV_process_no_space_removal_no_sanitiz:NNn {
6372   \KV_process_aux:NNNn \KV_parse_no_space_removal_no_sanitiz:n
6373 }
6374 \cs_new_protected:Npn \KV_process_aux:NNNn #1#2#3#4 {
6375   \cs_set_eq:cN
6376     { KV_key_no_value_elt_ \int_use:N \l_KV_level_int :n }
6377     \KV_key_no_value_elt:n
6378   \cs_set_eq:cN
6379     { KV_key_value_elt_ \int_use:N \l_KV_level_int :nn }
6380     \KV_key_value_elt:nn
6381   \cs_set_eq:NN \KV_key_no_value_elt:n #2
6382   \cs_set_eq:NN \KV_key_value_elt:nn #3
6383   \int_incr:N \l_KV_level_int
6384   #1 {#4}
6385   \int_decr:N \l_KV_level_int
```

```

6386 \cs_set_eq:Nc \KV_key_no_value_elt:n
6387 { KV_key_no_value_elt_ \int_use:N \l_KV_level_int :n }
6388 \cs_set_eq:Nc \KV_key_value_elt:nn
6389 { KV_key_value_elt_ \int_use:N \l_KV_level_int :nn }
6390 }

```

(End definition for \KV_process_space_removal_sanitization. This function is documented on page 133.)

\KV_sanitization_outerlevel_active_equals:N Some functions for sanitizing top level equals and commas. Replace =₁₃ and ,₁₃ with =₁₂ and ,₁₂ resp.

```

6391 \group_begin:
6392 \char_set_catcode:nn{'\}{13}
6393 \char_set_catcode:nn{'\,}{13}
6394 \char_set_lccode:nn{'\8}{=}
6395 \char_set_lccode:nn{'\9}{\,}
6396 \tl_to_lowercase:n{\group_end:
6397 \cs_new_protected_nopar:Npn \KV_sanitization_outerlevel_active_equals:N #1{
6398 \tl_replace_all_in:Nnn #1 = 8
6399 }
6400 \cs_new_nopar:Npn \KV_sanitization_outerlevel_active_commas:N #1{
6401 \tl_replace_all_in:Nnn #1 , 9
6402 }
6403 }

```

(End definition for \KV_sanitization_outerlevel_active_equals:N. This function is documented on page 385.)

\KV_remove_surrounding_spaces:nw
 \KV_remove_surrounding_spaces_auxi:w
 \KV_remove_surrounding_spaces_auxii:w
 \KV_set_key_element:w
 \KV_add_value_element:w

The macro \KV_remove_surrounding_spaces:nw removes a possible leading space plus a possible ending space from its second argument and stores it in the token register #1.

Based on Around the Bend No. 15 but with some enhancements. For instance, this definition is purely expandable.

We use a funny token Q₃ as a delimiter.

```

6404 \group_begin:
6405 \char_set_catcode:nn{'Q}{3}

6406 \cs_new:Npn \KV_remove_surrounding_spaces:nw#1#2\q_nil{

```

The idea in this processing is to use a Q with strange catcode to remove a trailing space. But first, how to get this expansion going?

If you have read the fine print in the l3expan module, you'll know that the f type expansion will expand until the first non-expandable token is seen and if this token is a space, it will be gobbled. Sounds useful for removing a leading space but we also need to make sure that it does nothing but removing that space! Therefore we prepend the argument to be trimmed with an \exp_not:N. Now why is that? \exp_not:N in itself is an expandable command so will allow the f expansion to continue. If the first token in the argument to

be trimmed is a space, it will be gobbled and the expansion stop. If the first token isn't a space, the `\exp_not:N` turns it temporarily into `\scan_stop:` which is unexpandable. The processing stops but the token following directly after `\exp_not:N` is now back to normal.

The function here allows you to insert arbitrary functions in the first argument but they should all be with an `f` type expansion. For the application in this module, we use `\tl_set:Nf`.

Once the expansion has been kick-started, we apply `\KV_remove_surrounding_spaces_auxi:w` to the replacement text of `#2`, adding a leading `\exp_not:N`. Note that no braces are stripped off of the original argument.

```
6407   #1{\KV_remove_surrounding_spaces_auxi:w \exp_not:N#2Q~Q}
6408 }
```

`\KV_remove_surrounding_spaces_auxi:w` removes a trailing space if present, then calls `\KV_remove_surrounding_spaces_auxii:w` to clean up any leftover bizarre Qs. In order for `\KV_remove_surrounding_spaces_auxii:w` to work properly we need to put back a Q first.

```
6409 \cs_new:Npn\KV_remove_surrounding_spaces_auxi:w#1~Q{
6410   \KV_remove_surrounding_spaces_auxii:w #1 Q
6411 }
```

Now all that is left to do is remove a leading space which should be taken care of by the function used to initiate the expansion. Simply return the argument before the funny Q.

```
6412 \cs_new:Npn\KV_remove_surrounding_spaces_auxii:w#1Q#2{#1}
```

Here are some specialized versions of the above. They do exactly what we want in one go. First trim spaces from the value and then put the result surrounded in braces onto `\l_KV_parse_tl`.

```
6413 \cs_new_protected:Npn\KV_add_value_element:w\q_stop#1\q_nil{
6414   \tl_set:Nf\l_KV_currval_tl {
6415     \KV_remove_surrounding_spaces_auxi:w \exp_not:N#1Q~Q
6416   }
6417   \tl_put_right:No\l_KV_parse_tl{
6418     \exp_after:wN { \l_KV_currval_tl }
6419   }
6420 }
```

When storing the key we firstly remove spaces plus the prepended `\q_no_value`.

```
6421 \cs_new_protected:Npn\KV_set_key_element:w#1\q_nil{
6422   \tl_set:Nf\l_KV_currkey_tl
6423   {
6424     \exp_last_unbraced:NNo \KV_remove_surrounding_spaces_auxi:w
6425     \exp_not:N \use_none:n #1Q~Q
6426   }
```

Afterwards we gobble an extra level of braces if that's what we are asked to do.

```

6427 \bool_if:NT \l_KV_remove_one_level_of_braces_bool
6428 {
6429   \exp_args:NNo \tl_set:No \l_KV_currkey_tl {
6430     \exp_after:wN \KV_add_element_aux:w \l_KV_currkey_tl \q_nil
6431   }
6432 }
6433 }
6434 \group_end:

```

(End definition for `\KV_remove_surrounding_spaces:nw`. This function is documented on page 385.)

`\KV_add_element_aux:w` A helper function for fixing braces around keys and values.

```

6435 \cs_new:Npn \KV_add_element_aux:w#1\q_nil{#1}

```

(End definition for `\KV_add_element_aux:w`.)

Parse a list of keyvals, put them into list form with entries like `\KV_key_no_value_elt:n{key1}` and `\KV_key_value_elt:nn{key2}{val2}`.

`\KV_parse_sanitizae_aux:n` The slow parsing algorithm sanitizes active commas and equal signs at the top level first. Then uses #1 as inspector of each element in the comma list.

```

6436 \cs_new_protected:Npn \KV_parse_sanitizae_aux:n #1 {
6437   \group_begin:
6438   \tl_clear:N \l_KV_parse_tl
6439   \tl_set:Nn \l_KV_tmpa_tl {#1}
6440   \KV_sanitizae_outerlevel_active_equals:N \l_KV_tmpa_tl
6441   \KV_sanitizae_outerlevel_active_commas:N \l_KV_tmpa_tl
6442   \exp_last_unbraced:NNV \KV_parse_elt:w \q_no_value
6443   \l_KV_tmpa_tl , \q_nil ,

```

We evaluate the parsed keys and values outside the group so the token register is restored to its previous value.

```

6444   \exp_after:wN \group_end:
6445   \l_KV_parse_tl
6446 }

```

(End definition for `\KV_parse_sanitizae_aux:n`.)

`\KV_parse_no_sanitizae_aux:n` Like above but we don't waste time sanitizing. This is probably the one we will use for preamble parsing where catcodes of = and , are as expected!

```

6447 \cs_new_protected:Npn \KV_parse_no_sanitizae_aux:n #1{
6448   \group_begin:
6449   \tl_clear:N \l_KV_parse_tl
6450   \KV_parse_elt:w \q_no_value #1 , \q_nil ,
6451   \exp_after:wN \group_end:
6452   \l_KV_parse_tl
6453 }

```

(End definition for `\KV_parse_no_sanitize_aux:n`.)

`\KV_parse_elt:w` This function will always have a `\q_no_value` stuffed in as the rightmost token in #1. In case there was a blank entry in the comma separated list we just run it again. The `\use_none:n` makes sure to gobble the quark `\q_no_value`. A similar test is made to check if we hit the end of the recursion.

```

6454 \cs_set:Npn \KV_parse_elt:w #1,{
6455   \tl_if_blank:oTF{\use_none:n #1}
6456   { \KV_parse_elt:w \q_no_value }
6457   {
6458     \quark_if_nil:oF {\use_ii:nn #1 }

```

If we made it to here we can start parsing the key and value. When done try, try again.

```

6459   {
6460     \KV_split_key_value_current:w #1==\q_nil
6461     \KV_parse_elt:w \q_no_value
6462   }
6463 }
6464 }

```

(End definition for `\KV_parse_elt:w`.)

`\KV_split_key_value_current:w` The function called to split the keys and values.

```

6465 \cs_new:Npn \KV_split_key_value_current:w {\ERROR}

```

(End definition for `\KV_split_key_value_current:w`. This function is documented on page 385.)

We provide two functions for splitting keys and values. The reason being that most of the time, we should probably be in the special coding regime where spaces are ignored. Hence it makes no sense to spend time searching for extra space tokens and we can do the settings directly. When comparing these two versions (neither doing any sanitizing) the `no_space_removal` version is more than 40% faster than `space_removal`.

It is up to functions like `\DeclareTemplate` to check which catcode regime is active and then pick up the version best suited for it.

`\KV_split_key_value_space_removal:w` The code below removes extraneous spaces around the keys and values plus one set of braces around the entire value.

`\KV_split_key_value_space_removal_detect_error:wTF`
`\KV_split_key_value_space_removal_aux:w`

Unlike the version to be used when spaces are ignored, this one only grabs the key which is everything up to the first = and save the rest for closer inspection. Reason is that if a user has entered `mykey={{myval}}`, then the outer braces have already been removed before we even look at what might come after the key. So this is slightly more tedious (but only slightly) but at least it always removes only one level of braces.

```

6466 \cs_new_protected:Npn \KV_split_key_value_space_removal:w #1 = #2\q_nil{

```

First grab the key.

```
6467 \KV_set_key_element:w#1\q_nil
```

Then we start checking. If only a key was entered, #2 contains = and nothing else, so we test for that first.

```
6468 \tl_set:Nn\l_KV_tmpa_tl{#2}
6469 \tl_if_eq:NNTF\l_KV_tmpa_tl\c_KV_single_equal_sign_tl
```

Then we just insert the default key.

```
6470 {
6471   \tl_put_right:No\l_KV_parse_tl{
6472     \exp_after:wN \KV_key_no_value_elt:n
6473     \exp_after:wN {\l_KV_currkey_tl}
6474   }
6475 }
```

Otherwise we must take a closer look at what is left. The remainder of the original list up to the comma is now stored in #2 plus an additional ==, which wasn't gobbled during the initial reading of arguments. If there is an error then we can see at least one more = so we call an auxiliary function to check for this.

```
6476 {
6477   \KV_split_key_value_space_removal_detect_error:wTF#2\q_no_value\q_nil
6478   {\KV_split_key_value_space_removal_aux:w \q_stop #2}
6479   {\msg_kernel_error:nn {keyval } {misplaced-equals-sign } }
6480 }
6481 }
```

The error test.

```
6482 \cs_new_protected:Npn
6483 \KV_split_key_value_space_removal_detect_error:wTF#1=#2#3\q_nil{
6484   \tl_if_head_eq_meaning:nNTF{#3}\q_no_value
6485 }
```

Now we can start extracting the value. Recall that #1 here starts with \q_stop so all braces are still there! First we try to see how much is left if we gobble three brace groups from #1. If #1 is empty or blank, all three quarks are gobbled. If #1 consists of exactly one token or brace group, only the latter quark is left.

```
6486 \cs_new:Npn \KV_val_preserve_braces:NnN #1#2#3{{#2}}
6487 \cs_new_protected:Npn\KV_split_key_value_space_removal_aux:w #1=={
6488   \tl_set:Nx\l_KV_tmpa_tl{\exp_not:o{\use_none:nnn#1\q_nil\q_nil}}
6489   \tl_put_right:No\l_KV_parse_tl{
6490     \exp_after:wN \KV_key_value_elt:nn
6491     \exp_after:wN {\l_KV_currkey_tl}
6492   }
```

If there a blank space or nothing at all, `\l_KV_tmpa_tl` is now completely empty.

```
6493 \tl_if_empty:NTF\l_KV_tmpa_tl
```

We just put an empty value on the stack.

```
6494 { \tl_put_right:Nn\l_KV_parse_tl{{}} }
6495 {
```

If there was exactly one brace group or token in #1, `\l_KV_tmpa_tl` is now equal to `\q_nil`. Then we can just pick it up as the second argument of #1. This will also take care of any spaces which might surround it.

```
6496 \quark_if_nil:NTF\l_KV_tmpa_tl
6497 {
6498   \bool_if:NTF \l_KV_remove_one_level_of_braces_bool
6499   {
6500     \tl_put_right:No\l_KV_parse_tl{
6501       \exp_after:wN{\use_ii:nnn #1\q_nil}
6502     }
6503   }
6504   {
6505     \tl_put_right:No\l_KV_parse_tl{
6506       \exp_after:wN{\KV_val_preserve_braces:NnN #1\q_nil}
6507     }
6508   }
6509 }
```

Otherwise we grab the value.

```
6510 { \KV_add_value_element:w #1\q_nil }
6511 }
6512 }
```

(End definition for `\KV_split_key_value_space_removal:w`. This function is documented on page 385.)

`\KV_split_key_value_no_space_removal:w` This version is for when in the special coding regime where spaces are ignored so there is no need to do any fancy space hacks, however fun they may be. Since there are no spaces, a set of braces around a value is automatically stripped by T_EX.

```
6513 \cs_new_protected:Npn \KV_split_key_value_no_space_removal:w #1#2=#3=#4\q_nil{
6514   \tl_set:Nn\l_KV_tmpa_tl{#4}
6515   \tl_if_empty:NTF \l_KV_tmpa_tl
6516   {
6517     \tl_put_right:Nn\l_KV_parse_tl{\KV_key_no_value_elt:n{#2}}
6518   }
6519   {
6520     \tl_if_eq:NNTF\c_KV_single_equal_sign_tl\l_KV_tmpa_tl
6521     {
6522       \tl_put_right:Nn\l_KV_parse_tl{\KV_key_value_elt:nn{#2}{#3}}
6523     }
6524   }
```

```

6524 { \msg_kernel_error:nn { keyval } { misplaced-equals-sign } }
6525 }
6526 }

```

(End definition for `\KV_split_key_value_no_space_removal:w`. This function is documented on page 385.)

`\KV_key_no_value_elt:n`
`\KV_key_value_elt:nn`

```

6527 \cs_new:Npn \KV_key_no_value_elt:n #1{\ERROR}
6528 \cs_new:Npn \KV_key_value_elt:nn #1#2{\ERROR}

```

(End definition for `\KV_key_no_value_elt:n`. This function is documented on page 134.)

`\KV_parse_no_space_removal_no_sanitize:n`

Finally we can put all the things together. `\KV_parse_no_space_removal_no_sanitize:n` is the version that disallows unmatched conditional and does no space removal.

```

6529 \cs_new_protected_nopar:Npn \KV_parse_no_space_removal_no_sanitize:n {
6530   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_no_space_removal:w
6531   \KV_parse_no_sanitize_aux:n
6532 }

```

(End definition for `\KV_parse_no_space_removal_no_sanitize:n`. This function is documented on page 133.)

`\KV_parse_space_removal_sanitize:n`
`\KV_parse_space_removal_no_sanitize:n`

The other varieties can be defined in a similar manner. For the version needed at the document level, we can use this one.

```

6537 \cs_new_protected_nopar:Npn \KV_parse_space_removal_sanitize:n {
6538   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_space_removal:w
6539   \KV_parse_sanitize_aux:n
6540 }

```

For preamble use by the non-programmer this is probably best.

```

6537 \cs_new_protected_nopar:Npn \KV_parse_space_removal_no_sanitize:n {
6538   \cs_set_eq:NN \KV_split_key_value_current:w \KV_split_key_value_space_removal:w
6539   \KV_parse_no_sanitize_aux:n
6540 }

```

(End definition for `\KV_parse_space_removal_sanitize:n`. This function is documented on page 134.)

```

6541 \msg_kernel_new:nnnn { keyval } { misplaced-equals-sign }
6542 {Misplaced-equals-sign-in-key--value-input~\msg_line_context:}
6543 {
6544   I~am~trying~to~read~some~key--value~input~but~found~two~equals~
6545   signs\\%
6546   without~a~comma~between~them.
6547 }

```

```

6548 </initex | package>

```


The usual preliminaries.

```
6549 <*package>
6550 \ProvidesExplPackage
6551   {\filename}{\filedate}{\fileversion}{\filedescription}
6552 \package_check_loaded_expl:
6553 </package>
6554 <*initex | package>
```

115.1.1 Variables and constants

`\c_keys_root_tl` Where the keys are really stored.
`\c_keys_properties_root_tl`

```
6555 \tl_const:Nn \c_keys_root_tl { keys~>~ }
6556 \tl_const:Nn \c_keys_properties_root_tl { keys_properties }
```

(End definition for `\c_keys_root_tl`. This function is documented on page 144.)

`\c_keys_value_forbidden_tl` Two marker token lists.
`\c_keys_value_required_tl`

```
6557 \tl_const:Nn \c_keys_value_forbidden_tl { forbidden }
6558 \tl_const:Nn \c_keys_value_required_tl { required }
```

(End definition for `\c_keys_value_forbidden_tl`. This function is documented on page 145.)

`\l_keys_choice_int` Used for the multiple choice system.
`\l_keys_choice_tl`

```
6559 \int_new:N \l_keys_choice_int
6560 \tl_new:N \l_keys_choice_tl
```

(End definition for `\l_keys_choice_int`. This function is documented on page 141.)

`\l_keys_choice_code_tl` When creating multiple choices, the code is stored here.

```
6561 \tl_new:N \l_keys_choice_code_tl
```

(End definition for `\l_keys_choice_code_tl`. This function is documented on page 145.)

`\l_keys_key_tl` Storage for the current key name and the path of the key (key name plus module name).
`\l_keys_path_tl`
`\l_keys_property_tl`

```
6562 \tl_new:N \l_keys_key_tl
6563 \tl_new:N \l_keys_path_tl
6564 \tl_new:N \l_keys_property_tl
```

(End definition for `\l_keys_key_tl`. This function is documented on page 145.)

`\l_keys_module_tl` The module for an entire set of keys.

```
6565 \tl_new:N \l_keys_module_tl
```

(End definition for `\l_keys_module_tl`. This function is documented on page 145.)

`\l_keys_no_value_bool` To indicate that no value has been given.

```
6566 \bool_new:N \l_keys_no_value_bool
```

(End definition for `\l_keys_no_value_bool`. This function is documented on page 145.)

`\l_keys_value_tl` A token variable for the given value.

```
6567 \tl_new:N \l_keys_value_tl
```

(End definition for `\l_keys_value_tl`. This function is documented on page 145.)

115.1.2 Internal functions

`\keys_bool_set:Nn` Boolean keys are really just choices, but all done by hand.

```
6568 \cs_new_protected_nopar:Npn \keys_bool_set:Nn #1#2 {  
6569   \keys_cmd_set:nx { \l_keys_path_tl / true } {  
6570     \exp_not:c { bool_ #2 set_true:N }  
6571     \exp_not:N #1  
6572   }  
6573   \keys_cmd_set:nx { \l_keys_path_tl / false } {  
6574     \exp_not:N \use:c  
6575     { bool_ #2 set_false:N }  
6576     \exp_not:N #1  
6577   }  
6578   \keys_choice_make:  
6579   \cs_if_exist:NF #1 {  
6580     \bool_new:N #1  
6581   }  
6582   \keys_default_set:n { true }  
6583 }
```

(End definition for `\keys_bool_set:Nn`. This function is documented on page 142.)

`\keys_choice_code_store:x` The code for making multiple choices is stored in a token list as there should not be any # tokens.

```
6584 \cs_new_protected:Npn \keys_choice_code_store:x #1 {  
6585   \tl_set:cx { \c_keys_root_tl \l_keys_path_tl .choice_code_tl } {#1}  
6586 }
```

(End definition for `\keys_choice_code_store:x`. This function is documented on page 142.)

`\keys_choice_find:n` Executing a choice has two parts. First, try the choice given, then if that fails call the unknown key. That will exist, as it is created when a choice is first made. So there is no need for any escape code.

```

6587 \cs_new_protected_nopar:Npn \keys_choice_find:n #1 {
6588   \keys_execute_aux:nn { \l_keys_path_tl / \tl_to_str:n {#1} } {
6589     \keys_execute_aux:nn { \l_keys_path_tl / unknown } { }
6590   }
6591 }

```

(End definition for `\keys_choice_find:n`. This function is documented on page 143.)

\keys_choice_make: To make a choice from a key, two steps: set the code, and set the unknown key.

```

6592 \cs_new_protected_nopar:Npn \keys_choice_make: {
6593   \keys_cmd_set:nn { \l_keys_path_tl } {
6594     \keys_choice_find:n {##1}
6595   }
6596   \keys_cmd_set:nn { \l_keys_path_tl / unknown } {
6597     \msg_kernel_error:nnxx { keys } { choice-unknown }
6598     { \l_keys_path_tl } {##1}
6599   }
6600 }

```

(End definition for `\keys_choice_make:`. This function is documented on page 142.)

\keys_choices_generate:n **\keys_choices_generate_aux:n** Creating multiple-choices means setting up the “indicator” code, then applying whatever the user wanted.

```

6601 \cs_new_protected:Npn \keys_choices_generate:n #1 {
6602   \keys_choice_make:
6603   \int_zero:N \l_keys_choice_int
6604   \cs_if_exist:cTF {
6605     \c_keys_root_tl \l_keys_path_tl .choice_code_tl
6606   } {
6607     \tl_set:Nv \l_keys_choice_code_tl {
6608       \c_keys_root_tl \l_keys_path_tl .choice_code_tl
6609     }
6610   }{
6611     \msg_kernel_error:nnx { keys } { generate-choices-before-code }
6612     { \l_keys_path_tl }
6613   }
6614   \clist_map_function:nN {#1} \keys_choices_generate_aux:n
6615 }
6616 \cs_new_protected_nopar:Npn \keys_choices_generate_aux:n #1 {
6617   \keys_cmd_set:nx { \l_keys_path_tl / #1 } {
6618     \exp_not:n { \tl_set:Nn \l_keys_choice_tl } {#1}
6619     \exp_not:n { \int_set:Nn \l_keys_choice_int }
6620     { \int_use:N \l_keys_choice_int }
6621     \exp_not:V \l_keys_choice_code_tl
6622   }
6623   \int_incr:N \l_keys_choice_int
6624 }

```

(End definition for `\keys_choices_generate:n`. This function is documented on page 142.)

`\keys_cmd_set:nn` Creating a new command means setting properties and then creating a function with the correct number of arguments.
`\keys_cmd_set:nx`

`\keys_cmd_set_aux:n`

```

6625 \cs_new_protected:Npn \keys_cmd_set:nn #1#2 {
6626   \keys_cmd_set_aux:n {#1}
6627   \cs_generate_from_arg_count:cNnn { \c_keys_root_tl #1 .cmd:n }
6628   \cs_set:Npn 1 {#2}
6629 }
6630 \cs_new_protected:Npn \keys_cmd_set:nx #1#2 {
6631   \keys_cmd_set_aux:n {#1}
6632   \cs_generate_from_arg_count:cNnn { \c_keys_root_tl #1 .cmd:n }
6633   \cs_set:Npx 1 {#2}
6634 }
6635 \cs_new_protected_nopar:Npn \keys_cmd_set_aux:n #1 {
6636   \keys_property_undefine:n { #1 .default_tl }
6637   \cs_if_free:cT { \c_keys_root_tl #1 .req_tl }
6638   { \tl_new:c { \c_keys_root_tl #1 .req_tl } }
6639   \tl_clear:c { \c_keys_root_tl #1 .req_tl }
6640 }

```

(End definition for `\keys_cmd_set:nn`. This function is documented on page 143.)

`\keys_default_set:n` Setting a default value is easy.
`\keys_default_set:V`

```

6641 \cs_new_protected:Npn \keys_default_set:n #1 {
6642   \cs_if_free:cT { \c_keys_root_tl \l_keys_path_tl .default_tl }
6643   { \tl_new:c { \c_keys_root_tl \l_keys_path_tl .default_tl } }
6644   \tl_set:cn { \c_keys_root_tl \l_keys_path_tl .default_tl } {#1}
6645 }
6646 \cs_generate_variant:Nn \keys_default_set:n { V }

```

(End definition for `\keys_default_set:n`. This function is documented on page 143.)

`\keys_define:nn` The main key-defining function mainly sets up things for `l3keyval` to use.
`\keys_define_aux:nnn`
`\keys_define_aux:onn`

```

6647 \cs_new_protected:Npn \keys_define:nn {
6648   \keys_define_aux:onn { \l_keys_module_tl }
6649 }
6650 \cs_new_protected:Npn \keys_define_aux:nnn #1#2#3 {
6651   \tl_set:Nn \l_keys_module_tl {#2}
6652   \KV_process_no_space_removal_no_sanitization:NNn
6653   \keys_define_elt:n \keys_define_elt:nn {#3}
6654   \tl_set:Nn \l_keys_module_tl {#1}
6655 }
6656 \cs_generate_variant:Nn \keys_define_aux:nnn { o }

```

(End definition for `\keys_define:nn`. This function is documented on page 136.)

`\keys_define_elt:n` The element processors for defining keys.
`\keys_define_elt:nn`

```

6657 \cs_new_protected_nopar:Npn \keys_define_elt:n #1 {
6658   \bool_set_true:N \l_keys_no_value_bool
6659   \keys_define_elt_aux:nn {#1} { }
6660 }
6661 \cs_new_protected:Npn \keys_define_elt:nn #1#2 {
6662   \bool_set_false:N \l_keys_no_value_bool
6663   \keys_define_elt_aux:nn {#1} {#2}
6664 }

```

(End definition for `\keys_define_elt:n`. This function is documented on page 143.)

`\keys_define_elt_aux:nn` The auxiliary function does most of the work.

```

6665 \cs_new_protected:Npn \keys_define_elt_aux:nn #1#2 {
6666   \keys_property_find:n {#1}
6667   \cs_set_eq:Nc \keys_tmp:w
6668   { \c_keys_properties_root_tl \l_keys_property_tl }
6669   \cs_if_exist:NTF \keys_tmp:w {
6670     \keys_define_key:n {#2}
6671   }{
6672     \msg_kernel_error:nnxx { keys } { property-unknown }
6673     { \l_keys_property_tl } { \l_keys_path_tl }
6674   }
6675 }

```

(End definition for `\keys_define_elt_aux:nn`.)

`\keys_define_key:n` Defining a new key means finding the code for the appropriate property then running it. As properties have signatures, a check can be made for required values without needing anything set explicitly.

```

6676 \cs_new_protected:Npn \keys_define_key:n #1 {
6677   \bool_if:NTF \l_keys_no_value_bool {
6678     \int_compare:nTF {
6679       \exp_args:Nc \cs_get_arg_count_from_signature:N
6680       { \l_keys_property_tl } = \c_zero
6681     } {
6682       \keys_tmp:w
6683     }{
6684       \msg_kernel_error:nnxx { key } { property-requires-value }
6685       { \l_keys_property_tl } { \l_keys_path_tl }
6686     }
6687   }{
6688     \keys_tmp:w {#1}
6689   }
6690 }

```

(End definition for `\keys_define_key:n`. This function is documented on page 143.)

`\keys_execute:` Actually executing a key is done in two parts. First, look for the key itself, then look for
`\keys_execute_unknown:` the `unknown` key with the same path. If both of these fail, complain!
`\keys_execute_aux:nn`

```

6691 \cs_new_protected_nopar:Npn \keys_execute: {
6692   \keys_execute_aux:nn { \l_keys_path_tl } {
6693     \keys_execute_unknown:
6694   }
6695 }
6696 \cs_new_protected_nopar:Npn \keys_execute_unknown: {
6697   \keys_execute_aux:nn { \l_keys_module_tl / unknown } {
6698     \msg_kernel_error:nxxx { keys } { key-unknown } { \l_keys_path_tl }
6699     { \l_keys_module_tl }
6700   }
6701 }

```

If there is only one argument required, it is wrapped in braces so that everything is passed through properly. On the other hand, if more than one is needed it is down to the user to have put things in correctly! The use of `\q_keys_stop` here means that arguments do not run away (hence the nine empty groups), but that the module can clean up the spare groups at the end of executing the key.

```

6702 \cs_new_protected_nopar:Npn \keys_execute_aux:nn #1#2 {
6703   \cs_set_eq:Nc \keys_tmp:w { \c_keys_root_tl #1 .cmd:n }
6704   \cs_if_exist:NTF \keys_tmp:w {
6705     \exp_args:NV \keys_tmp:w \l_keys_value_tl
6706   }{
6707     #2
6708   }
6709 }

```

(End definition for `\keys_execute:`. This function is documented on page 143.)

`\keys_if_exist:nnTF` A check for the existence of a key. This works by looking for the command function for the key (which ends `.cmd:n`).

```

6710 \prg_set_conditional:Nnn \keys_if_exist:nn {TF,T,F} {
6711   \cs_if_exist:cTF { \c_keys_root_tl #1 / #2 .cmd:n } {
6712     \prg_return_true:
6713   }{
6714     \prg_return_false:
6715   }
6716 }

```

(End definition for `\keys_if_exist:nn`. This function is documented on page 142.)

`\keys_if_value_requirement:nTF` To test if a value is required or forbidden. Only one version is needed, so done by hand.

```

6717 \cs_new_nopar:Npn \keys_if_value_requirement:nTF #1 {
6718   \tl_if_eq:ccTF { c_keys_value_ #1 _tl } {
6719     \c_keys_root_tl \l_keys_path_tl .req_tl
6720   }
6721 }

```

(End definition for `\keys_if_value_requirement:nTF`. This function is documented on page 143.)

`\keys_meta_make:n` To create a met-key, simply set up to pass data through.

`\keys_meta_make:x`

```
6722 \cs_new_protected_nopar:Npn \keys_meta_make:n #1 {  
6723   \exp_last_unbraced:NNo \keys_cmd_set:nn \l_keys_path_tl  
6724     \exp_after:wN { \exp_after:wN \keys_set:nn \exp_after:wN { \l_keys_module_tl } {#1} }  
6725 }  
6726 \cs_new_protected_nopar:Npn \keys_meta_make:x #1 {  
6727   \keys_cmd_set:nx { \l_keys_path_tl } {  
6728     \exp_not:N \keys_set:nn { \l_keys_module_tl } {#1}  
6729   }  
6730 }
```

(End definition for `\keys_meta_make:n`. This function is documented on page 143.)

`\keys_property_find:n` Searching for a property means finding the last “.” in the input, and storing the text before and after it.

`\keys_property_find_aux:n`

`\keys_property_find_aux:w`

```
6731 \cs_new_protected_nopar:Npn \keys_property_find:n #1 {  
6732   \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / }  
6733   \tl_if_in:nnTF {#1} {.} {  
6734     \keys_property_find_aux:n {#1}  
6735   }{  
6736     \msg_kernel_error:nnx { keys } { key-no-property } {#1}  
6737   }  
6738 }  
6739 \cs_new_protected_nopar:Npn \keys_property_find_aux:n #1 {  
6740   \keys_property_find_aux:w #1 \q_stop  
6741 }  
6742 \cs_new_protected_nopar:Npn \keys_property_find_aux:w #1 . #2 \q_stop {  
6743   \tl_if_in:nnTF {#2} { . } {  
6744     \tl_set:Nx \l_keys_path_tl {  
6745       \l_keys_path_tl \tl_to_str:n {#1} .  
6746     }  
6747     \keys_property_find_aux:w #2 \q_stop  
6748   }{  
6749     \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }  
6750     \tl_set:Nn \l_keys_property_tl { . #2 }  
6751   }  
6752 }
```

(End definition for `\keys_property_find:n`. This function is documented on page 143.)

`\keys_property_new:nn` Creating a new property is simply a case of making the correctly-named function.

`\keys_property_new_arg:nn`

```
6753 \cs_new_nopar:Npn \keys_property_new:nn #1#2 {  
6754   \cs_new:cpn { \c_keys_properties_root_tl #1 } {#2}  
6755 }  
6756 \cs_new_protected_nopar:Npn \keys_property_new_arg:nn #1#2 {  
6757   \cs_new:cpn { \c_keys_properties_root_tl #1 } ##1 {#2}  
6758 }
```

(End definition for `\keys_property_new:nn`. This function is documented on page 144.)

`\keys_property_undefine:n` Removing a property means undefining it.

```
6759 \cs_new_protected_nopar:Npn \keys_property_undefine:n #1 {  
6760   \cs_set_eq:cN { \c_keys_root_tl #1 } \c_undefined  
6761 }
```

(End definition for `\keys_property_undefine:n`. This function is documented on page 144.)

`\keys_set:nn` The main setting function just does the set up to get `l3keyval` to do the hard work.
`\keys_set:nV`
`\keys_set:nv`
`\keys_set_aux:nnn`
`\keys_set_aux:onn`

```
6762 \cs_new_protected:Npn \keys_set:nn {  
6763   \keys_set_aux:onn { \l_keys_module_tl }  
6764 }  
6765 \cs_generate_variant:Nn \keys_set:nn { nV, nv }  
6766 \cs_new_protected:Npn \keys_set_aux:nnn #1#2#3 {  
6767   \tl_set:Nn \l_keys_module_tl {#2}  
6768   \KV_process_space_removal_sanitiz:Nn  
6769     \keys_set_elt:n \keys_set_elt:nn {#3}  
6770   \tl_set:Nn \l_keys_module_tl {#1}  
6771 }  
6772 \cs_generate_variant:Nn \keys_set_aux:nnn { o }
```

(End definition for `\keys_set:nn`. This function is documented on page ??.)

`\keys_set_elt:n` The two element processors are almost identical, and pass the data through to the underlying auxiliary, which does the work.
`\keys_set_elt:nn`

```
6773 \cs_new_protected_nopar:Npn \keys_set_elt:n #1 {  
6774   \bool_set_true:N \l_keys_no_value_bool  
6775   \keys_set_elt_aux:nn {#1} { }  
6776 }  
6777 \cs_new_protected:Npn \keys_set_elt:nn #1#2 {  
6778   \bool_set_false:N \l_keys_no_value_bool  
6779   \keys_set_elt_aux:nn {#1} {#2}  
6780 }
```

(End definition for `\keys_set_elt:n`. This function is documented on page 144.)

`\keys_set_elt_aux:nn` First, set the current path and add a default if needed. There are then checks to see if
`\keys_set_elt_aux:` the a value is required or forbidden. If everything passes, move on to execute the code.

```
6781 \cs_new_protected:Npn \keys_set_elt_aux:nn #1#2 {  
6782   \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }  
6783   \tl_set:Nx \l_keys_path_tl { \l_keys_module_tl / \l_keys_key_tl }  
6784   \keys_value_or_default:n {#2}  
6785   \keys_if_value_requirement:nTF { required } {  
6786     \bool_if:NTF \l_keys_no_value_bool {  
6787       \msg_kernel_error:nnx { keys } { value-required }  
6788     }  
6789   }  
6790 }
```



```

6788         { \l_keys_path_tl }
6789     }{
6790         \keys_set_elt_aux:
6791     }
6792 }{
6793     \keys_set_elt_aux:
6794 }
6795 }
6796 \cs_new_protected_nopar:Npn \keys_set_elt_aux: {
6797     \keys_if_value_requirement:nTF { forbidden } {
6798         \bool_if:NTF \l_keys_no_value_bool {
6799             \keys_execute:
6800         }{
6801             \msg_kernel_error:nnxx { keys } { value-forbidden }
6802             { \l_keys_path_tl } { \tl_use:N \l_keys_value_tl }
6803         }
6804     }{
6805         \keys_execute:
6806     }
6807 }

```

(End definition for `\keys_set_elt_aux:nn`.)

\keys_show:nn Showing a key is just a question of using the correct name.

```

6808 \cs_new_nopar:Npn \keys_show:nn #1#2 {
6809     \cs_show:c { \c_keys_root_tl #1 / \tl_to_str:n {#2} .cmd:n }
6810 }

```

(End definition for `\keys_show:nn`. This function is documented on page 142.)

\keys_tmp:w This scratch function is used to actually execute keys.

```

6811 \cs_new:Npn \keys_tmp:w {}

```

(End definition for `\keys_tmp:w`. This function is documented on page 144.)

\keys_value_or_default:n If a value is given, return it as #1, otherwise send a default if available.

```

6812 \cs_new_protected:Npn \keys_value_or_default:n #1 {
6813     \tl_set:Nn \l_keys_value_tl {#1}
6814     \bool_if:NT \l_keys_no_value_bool {
6815         \cs_if_exist:cT { \c_keys_root_tl \l_keys_path_tl .default_tl } {
6816             \tl_set:Nv \l_keys_value_tl {
6817                 \c_keys_root_tl \l_keys_path_tl .default_tl
6818             }
6819         }
6820     }
6821 }

```

(End definition for `\keys_value_or_default:n`. This function is documented on page 144.)

\keys_value_requirement:n Values can be required or forbidden by having the appropriate marker set.

```

6822 \cs_new_protected_nopar:Npn \keys_value_requirement:n #1 {
6823   \tl_set_eq:cc { \c_keys_root_tl \l_keys_path_tl .req_tl }
6824   { c_keys_value_ #1 _tl }
6825 }

```

(End definition for `\keys_value_requirement:n`. This function is documented on page 144.)

\keys_variable_set:NnNN **\keys_variable_set:cnNN** Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

```

6826 \cs_new_protected_nopar:Npn \keys_variable_set:NnNN #1#2#3#4 {
6827   \cs_if_exist:NF #1 {
6828     \use:c { #2 _new:N } #1
6829   }
6830   \keys_cmd_set:nx { \l_keys_path_tl } {
6831     \exp_not:c { #2 _ #3 set:N #4 } \exp_not:N #1 {##1}
6832   }
6833 }
6834 \cs_generate_variant:Nn \keys_variable_set:NnNN { c }

```

(End definition for `\keys_variable_set:NnNN`. This function is documented on page 144.)

115.1.3 Properties

.bool_set:N One function for this.

.bool_gset:N

```

6835 \keys_property_new_arg:nn { .bool_set:N } {
6836   \keys_bool_set:Nn #1 { }
6837 }
6838 \keys_property_new_arg:nn { .bool_gset:N } {
6839   \keys_bool_set:Nn #1 { g }
6840 }

```

(End definition for `.bool_set:N`. This function is documented on page 137.)

.choice: Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```

6841 \keys_property_new:nn { .choice: } {
6842   \keys_choice_make:
6843 }

```

(End definition for `.choice:.` This function is documented on page 137.)

.choice_code:n **.choice_code:x** Storing the code for choices, using `\exp_not:n` to avoid needing two internal functions.

```

6844 \keys_property_new_arg:nn { .choice_code:n } {
6845   \keys_choice_code_store:x { \exp_not:n {#1} }
6846 }
6847 \keys_property_new_arg:nn { .choice_code:x } {
6848   \keys_choice_code_store:x {#1}
6849 }

```

(End definition for `.choice_code:n`. This function is documented on page 137.)

`.code:n` Creating code is simply a case of passing through to the underlying `set` function.

```
.code:x
6850 \keys_property_new_arg:nn { .code:n } {
6851   \keys_cmd_set:nn { \l_keys_path_tl } {#1}
6852 }
6853 \keys_property_new_arg:nn { .code:x } {
6854   \keys_cmd_set:nx { \l_keys_path_tl } {#1}
6855 }
```

(End definition for `.code:n`. This function is documented on page 137.)

`.default:n` Expansion is left to the internal functions.

```
.default:V
6856 \keys_property_new_arg:nn { .default:n } {
6857   \keys_default_set:n {#1}
6858 }
6859 \keys_property_new_arg:nn { .default:V } {
6860   \keys_default_set:V #1
6861 }
```

(End definition for `.default:n`. This function is documented on page 137.)

`.dim_set:N` Setting a variable is very easy: just pass the data along.

```
.dim_set:c
.dim_gset:N
.dim_gset:c
6862 \keys_property_new_arg:nn { .dim_set:N } {
6863   \keys_variable_set:NnNN #1 { dim } { } n
6864 }
6865 \keys_property_new_arg:nn { .dim_set:c } {
6866   \keys_variable_set:cnNN {#1} { dim } { } n
6867 }
6868 \keys_property_new_arg:nn { .dim_gset:N } {
6869   \keys_variable_set:NnNN #1 { dim } g n
6870 }
6871 \keys_property_new_arg:nn { .dim_gset:c } {
6872   \keys_variable_set:cnNN {#1} { dim } g n
6873 }
```

(End definition for `.dim_set:N`. This function is documented on page 138.)

`.fp_set:N` Setting a variable is very easy: just pass the data along.

```
.fp_set:c
.fp_gset:N
.fp_gset:c
6874 \keys_property_new_arg:nn { .fp_set:N } {
6875   \keys_variable_set:NnNN #1 { fp } { } n
6876 }
6877 \keys_property_new_arg:nn { .fp_set:c } {
6878   \keys_variable_set:cnNN {#1} { fp } { } n
6879 }
6880 \keys_property_new_arg:nn { .fp_gset:N } {
```

```

6881 \keys_variable_set:NnNN #1 { fp } g n
6882 }
6883 \keys_property_new_arg:nn { .fp_gset:c } {
6884 \keys_variable_set:cnNN {#1} { fp } g n
6885 }

```

(End definition for `.fp_set:N`. This function is documented on page 138.)

`.generate_choices:n` Making choices is easy.

```

6886 \keys_property_new_arg:nn { .generate_choices:n } {
6887 \keys_choices_generate:n {#1}
6888 }

```

(End definition for `.generate_choices:n`. This function is documented on page 138.)

`.int_set:N` Setting a variable is very easy: just pass the data along.

```

.int_set:c
.int_gset:N
.int_gset:c
6889 \keys_property_new_arg:nn { .int_set:N } {
6890 \keys_variable_set:NnNN #1 { int } { } n
6891 }
6892 \keys_property_new_arg:nn { .int_set:c } {
6893 \keys_variable_set:cnNN {#1} { int } { } n
6894 }
6895 \keys_property_new_arg:nn { .int_gset:N } {
6896 \keys_variable_set:NnNN #1 { int } g n
6897 }
6898 \keys_property_new_arg:nn { .int_gset:c } {
6899 \keys_variable_set:cnNN {#1} { int } g n
6900 }

```

(End definition for `.int_set:N`. This function is documented on page 138.)

`.meta:n` Making a meta is handled internally.

```

.meta:x
6901 \keys_property_new_arg:nn { .meta:n } {
6902 \keys_meta_make:n {#1}
6903 }
6904 \keys_property_new_arg:nn { .meta:x } {
6905 \keys_meta_make:x {#1}
6906 }

```

(End definition for `.meta:n`. This function is documented on page 139.)

`.skip_set:N` Setting a variable is very easy: just pass the data along.

```

.skip_set:c
.skip_gset:N
.skip_gset:c
6907 \keys_property_new_arg:nn { .skip_set:N } {
6908 \keys_variable_set:NnNN #1 { skip } { } n
6909 }
6910 \keys_property_new_arg:nn { .skip_set:c } {

```

```

6911 \keys_variable_set:cnNN {#1} { skip } { } n
6912 }
6913 \keys_property_new_arg:nn { .skip_gset:N } {
6914 \keys_variable_set:NnNN #1 { skip } g n
6915 }
6916 \keys_property_new_arg:nn { .skip_gset:c } {
6917 \keys_variable_set:cnNN {#1} { skip } g n
6918 }

```

(End definition for `.skip_set:N`. This function is documented on page 139.)

```

.tl_set:N Setting a variable is very easy: just pass the data along.
.tl_set:c
.tl_set_x:N
.tl_set_x:c
.tl_gset:N
.tl_gset:c
.tl_gset_x:N
.tl_gset_x:c
6919 \keys_property_new_arg:nn { .tl_set:N } {
6920 \keys_variable_set:NnNN #1 { tl } { } n
6921 }
6922 \keys_property_new_arg:nn { .tl_set:c } {
6923 \keys_variable_set:cnNN {#1} { tl } { } n
6924 }
6925 \keys_property_new_arg:nn { .tl_set_x:N } {
6926 \keys_variable_set:NnNN #1 { tl } { } x
6927 }
6928 \keys_property_new_arg:nn { .tl_set_x:c } {
6929 \keys_variable_set:cnNN {#1} { tl } { } x
6930 }
6931 \keys_property_new_arg:nn { .tl_gset:N } {
6932 \keys_variable_set:NnNN #1 { tl } g n
6933 }
6934 \keys_property_new_arg:nn { .tl_gset:c } {
6935 \keys_variable_set:cnNN {#1} { tl } g n
6936 }
6937 \keys_property_new_arg:nn { .tl_gset_x:N } {
6938 \keys_variable_set:NnNN #1 { tl } g x
6939 }
6940 \keys_property_new_arg:nn { .tl_gset_x:c } {
6941 \keys_variable_set:cnNN {#1} { tl } g x
6942 }

```

(End definition for `.tl_set:N`. This function is documented on page 139.)

```

.value_forbidden: These are very similar, so both call the same function.
.value_required:
6943 \keys_property_new:nn { .value_forbidden: } {
6944 \keys_value_requirement:n { forbidden }
6945 }
6946 \keys_property_new:nn { .value_required: } {
6947 \keys_value_requirement:n { required }
6948 }

```

(End definition for `.value_forbidden:.` This function is documented on page 139.)

115.1.4 Messages

For when there is a need to complain.

```
6949 \msg_kernel_new:nnnn { keys } { choice-unknown }
6950 { Choice~'#2'~unknown~for~key~'#1'. }
6951 {
6952   The~key~'#1'~takes~a~limited~number~of~values.\\
6953   The~input~given,~'#2',~is~not~on~the~list~accepted.
6954 }
6955 \msg_kernel_new:nnnn { keys } { generate-choices-before-code }
6956 { No~code~available~to~generate~choices~for~key~'#1'. }
6957 {
6958   \l_msg_coding_error_text_tl
6959   Before~using~,generate_choices:n~the~code~should~be~defined\\%
6960   with~,choice_code:n~or~,choice_code:x.
6961 }
6962 \msg_kernel_new:nnnn { keys } { key-no-property }
6963 { No~property~given~in~definition~of~key~'#1'. }
6964 {
6965   \c_msg_coding_error_text_tl
6966   Inside~\token_to_str:N \keys_define:nn \c_space_tl each~key~name
6967   needs~a~property: \\
6968   ~ ~ #1 .<property> \\
6969   LaTeX~did~not~find~a~'.'~to~indicate~the~start~of~a~property.
6970 }
6971 \msg_kernel_new:nnnn { keys } { key-unknown }
6972 { The~key~'#1'~is~unknown~and~is~being~ignored. }
6973 {
6974   The~module~'#2'~does~not~have~a~key~called~'#1'.\\
6975   Check~that~you~have~spelled~the~key~name~correctly.
6976 }
6977 \msg_kernel_new:nnnn { keys } { property-requires-value }
6978 { The~property~'#1'~requires~a~value. }
6979 {
6980   \l_msg_coding_error_text_tl
6981   LaTeX~was~asked~to~set~property~'#2'~for~key~'#1'.\\
6982   No~value~was~given~for~the~property,~and~one~is~required.
6983 }
6984 \msg_kernel_new:nnnn { keys } { property-unknown }
6985 { The~key~property~'#1'~is~unknown. }
6986 {
6987   \l_msg_coding_error_text_tl
6988   LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':\\
6989   this~property~is~not~defined.
6990 }
6991 \msg_kernel_new:nnnn { keys } { value-forbidden }
6992 { The~key~'#1'~does~not~taken~a~value. }
6993 {
6994   The~key~'#1'~should~be~given~without~a~value.\\
```

```

6995     LaTeX~will~ignore~the~given~value~'#2'.
6996   }
6997   \msg_kernel_new:nnnn { keys } { value-required }
6998   { The~key~'#1'~requires~a~value. }
6999   {
7000     The~key~'#1'~must~have~a~value.\\
7001     No~value~was~present:~the~key~will~be~ignored.
7002   }

7003 </initex | package>

```

116 Internal file functions

`\g_file_stack_seq` Stores the stack of nested files loaded using `\file_input:n`. This is needed to restore the appropriate file name to `\g_file_current_name_tl` at the end of each file.

`\g_file_record_seq` Stores the name of every file loaded using `\file_input:n`. In contrast to `\g_file_stack_seq`, no items are ever removed from this sequence.

`\l_file_name_tl` Used to return the full name of a file for internal use.

`\l_file_search_path_seq` The sequence of file paths to search when loading a file.

`\l_file_search_path_saved_seq` When loaded on top of L^AT_EX 2_ε, there is a need to save the search path so that `\input@path` can be used as appropriate.

117 File operation implementation

The following test files are used for this code: m3file001.

```

7004 <*initex | package>

7005 <*package>
7006 \ProvidesExplPackage
7007   {\filename}{\filedate}{\fileversion}{\filedescription}
7008 \package_check_loaded_expl:
7009 </package>

```

`\g_file_current_name_tl` The name of the current file should be available at all times.

```

7010 \tl_new:N \g_file_current_name_tl

```

For the format the file name needs to be picked up at the start of the file. In package mode the current file name is collected from L^AT_EX 2_ε.

```

7011 <*initex>
7012 \tex_everyjob:D \exp_after:wN
7013 {
7014   \tex_the:D \tex_everyjob:D
7015   \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
7016 }
7017 </initex>
7018 <*package>
7019 \tl_gset_eq:NN \g_file_current_name_tl \@currname
7020 </package>

```

(End definition for `\g_file_current_name_tl`. This function is documented on page 145.)

`\g_file_stack_seq` The input list of files is stored as a sequence stack.

```

7021 \seq_new:N \g_file_stack_seq

```

(End definition for `\g_file_stack_seq`. This function is documented on page 408.)

`\g_file_record_seq` The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list.

```

7022 \seq_new:N \g_file_record_seq

```

The current file name should be included in the file list!

```

7023 <*initex>
7024 \tex_everyjob:D \exp_after:wN
7025 {
7026   \tex_the:D \tex_everyjob:D
7027   \seq_gput_right:NV \g_file_record_seq \g_file_current_name_tl
7028 }
7029 </initex>

```

(End definition for `\g_file_record_seq`. This function is documented on page 408.)

`\l_file_name_tl` Used to return the fully-qualified name of a file.

```

7030 \tl_new:N \l_file_name_tl

```

(End definition for `\l_file_name_tl`. This function is documented on page 408.)

`\l_file_search_path_seq` The current search path.

```

7031 \seq_new:N \l_file_search_path_seq

```

(End definition for `\l_file_search_path_seq`. This function is documented on page 408.)

`\l_file_search_path_saved_seq` The current search path has to be saved for package use.

```

7032 <*package>
7033 \seq_new:N \l_file_search_path_saved_seq
7034 </package>

```

(End definition for `\l_file_search_path_saved_seq`. This function is documented on page 408.)

`\file_add_path:nN` The way to test if a file exists is to try to open it: if it does not exist then T_EX will report end-of-file. For files which are in the current directory, this is straight-forward.
`\g_file_test_stream` For other locations, a search has to be made looking at each potential path in turn. The first location is of course treated as the correct one. If nothing is found, #2 is returned empty.
`\file_add_path_search:nN`

```

7035 \cs_new_protected_nopar:Npn \file_add_path:nN #1#2
7036 {
7037   \ior_open:Nn \g_file_test_stream {#1}
7038   \ior_if_eof:NTF \g_file_test_stream
7039     { \file_add_path_search:nN {#1} #2 }
7040   {
7041     \ior_close:N \g_file_test_stream
7042     \tl_set:Nx #2 {#1}
7043   }
7044 }
7045 \cs_new_protected_nopar:Npn \file_add_path_search:nN #1#2
7046 {
7047   \tl_clear:N #2
7048   <*package>
7049   \cs_if_exist:NT \input@path
7050   {
7051     \seq_set_eq:NN \l_file_search_path_saved_seq \l_file_search_path_seq
7052     \clist_map_inline:Nn \input@path
7053       { \seq_put_right:Nn \l_file_search_path_seq {##1} }
7054   }
7055   </package>
7056   \seq_map_inline:Nn \l_file_search_path_seq
7057   {
7058     \ior_open:Nn \g_file_test_stream { ##1 #1 }
7059     \ior_if_eof:NF \g_file_test_stream
7060     {
7061       \tl_set:Nx #2 { ##1 #1 }
7062       \seq_map_break:
7063     }
7064   }
7065   <*package>
7066   \cs_if_exist:NT \input@path
7067   { \seq_set_eq:NN \l_file_search_path_seq \l_file_search_path_saved_seq }
7068   </package>
7069   \ior_close:N \g_file_test_stream
7070 }

```

(End definition for `\file_add_path:nN`. This function is documented on page 146.)

`\file_if_exist:nTF` The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path will contain something, whereas if the file was not located then the return value will be empty.

```

7071 \prg_new_protected_conditional:Nnn \file_if_exist:n { T , F , TF }
7072 {
7073   \file_add_path:nN {#1} \l_file_name_tl
7074   \tl_if_empty:NTF \l_file_name_tl
7075     { \prg_return_false: }
7076     { \prg_return_true: }
7077 }

```

(End definition for `\file_if_exist:n`. This function is documented on page 145.)

`\file_input:n` Loading a file is done in a safe way, checking first that the file exists and loading only if it does.

```

7078 \cs_new_protected_nopar:Npn \file_input:n #1
7079 {
7080   \file_add_path:nN {#1} \l_file_name_tl
7081   \tl_if_empty:NF \l_file_name_tl
7082   {
7083     <*initex>
7084     \seq_gput_right:Nx \g_file_record_seq {#1}
7085     </initex>
7086     <*package>
7087     \@addtofilelist {#1}
7088     </package>
7089     \seq_gpush:NV \g_file_stack_seq \g_file_current_name_tl
7090     \tl_gset:Nn \g_file_current_name_tl {#1}
7091     \tex_expandafter:D \tex_input:D \l_file_name_tl ~
7092     \seq_gpop:NN \g_file_stack_seq \g_file_current_name_tl
7093   }
7094 }

```

(End definition for `\file_input:n`. This function is documented on page 146.)

`\file_path_include:n` Wrapper functions to manage the search path.
`\file_path_remove:n`

```

7095 \cs_new_protected_nopar:Npn \file_path_include:n #1
7096 {
7097   \seq_if_in:NnF \l_file_search_path_seq {#1}
7098     { \seq_put_right:Nn \l_file_search_path_seq {#1} }
7099 }
7100 \cs_new_protected_nopar:Npn \file_path_remove:n #1
7101 { \seq_remove_all:Nn \l_file_search_path_seq {#1} }

```

(End definition for `\file_path_include:n`. This function is documented on page 146.)

`\file_list:` A function to list all files used to the log.

```

7102 \cs_new_protected_nopar:Npn \file_list:
7103 {
7104   \seq_remove_duplicates:N \g_file_record_seq
7105   \iow_log:n { *~File~List~* }
7106   \seq_map_inline:Nn \g_file_record_seq { \iow_log:n {##1} }
7107   \iow_log:n { ***** }
7108 }

```

(End definition for `\file_list:`. This function is documented on page 146.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here.

```

7109 <*package>
7110 \AtBeginDocument
7111 {
7112   \clist_map_inline:Nn \@filelist
7113   { \seq_put_right:Nn \g_file_record_seq {#1} }
7114 }
7115 </package>

7116 </initex | package>

```

118 Implementation

The following test files are used for this code: `m3fp003.lvt`.

We start by ensuring that the required packages are loaded.

```

7117 <*package>
7118 \ProvidesExplPackage
7119   {\filename}{\filedate}{\fileversion}{\filedescription}
7120 \package_check_loaded_expl:
7121 </package>
7122 <*initex | package>

```

118.1 Constants

<pre> \c_forty_four \c_one_hundred \c_one_thousand \c_one_million \c_one_hundred_million \c_five_hundred_million \c_one_thousand_million </pre>	<p>There is some speed to gain by moving numbers into fixed positions.</p> <pre> 7123 \int_const:Nn \c_forty_four { 44 } 7124 \int_const:Nn \c_one_hundred { 100 } 7125 \int_const:Nn \c_one_thousand { 1000 } 7126 \int_const:Nn \c_one_million { 1 000 000 } 7127 \int_const:Nn \c_one_hundred_million { 100 000 000 } 7128 \int_const:Nn \c_five_hundred_million { 500 000 000 } 7129 \int_const:Nn \c_one_thousand_million { 1 000 000 000 } </pre>
---	---

(End definition for `\c_forty_four`.)

`\c_fp_pi_by_four_decimal_int` Parts of π for trigonometric range reduction, implemented as `int` variables for speed.
`\c_fp_pi_by_four_extended_int`
`\c_fp_pi_decimal_int`
`\c_fp_pi_extended_int`
`\c_fp_two_pi_decimal_int`
`\c_fp_two_pi_extended_int`

```

7130 \int_new:N \c_fp_pi_by_four_decimal_int
7131 \int_set:Nn \c_fp_pi_by_four_decimal_int { 785 398 158 }
7132 \int_new:N \c_fp_pi_by_four_extended_int
7133 \int_set:Nn \c_fp_pi_by_four_extended_int { 897 448 310 }
7134 \int_new:N \c_fp_pi_decimal_int
7135 \int_set:Nn \c_fp_pi_decimal_int { 141 592 653 }
7136 \int_new:N \c_fp_pi_extended_int
7137 \int_set:Nn \c_fp_pi_extended_int { 589 793 238 }
7138 \int_new:N \c_fp_two_pi_decimal_int
7139 \int_set:Nn \c_fp_two_pi_decimal_int { 283 185 307 }
7140 \int_new:N \c_fp_two_pi_extended_int
7141 \int_set:Nn \c_fp_two_pi_extended_int { 179 586 477 }

```

(End definition for `\c_fp_pi_by_four_decimal_int`.)

`\c_e_fp` The value e as a ‘machine number’.

```

7142 \tl_new:N \c_e_fp
7143 \tl_set:Nn \c_e_fp { + 2.718281828 e 0 }

```

(End definition for `\c_e_fp`. This function is documented on page 147.)

`\c_one_fp` The constant value 1: used for fast comparisons.

```

7144 \tl_new:N \c_one_fp
7145 \tl_set:Nn \c_one_fp { + 1.000000000 e 0 }

```

(End definition for `\c_one_fp`. This function is documented on page 147.)

`\c_pi_fp` The value π as a ‘machine number’.

```

7146 \tl_new:N \c_pi_fp
7147 \tl_set:Nn \c_pi_fp { + 3.141592654 e 0 }

```

(End definition for `\c_pi_fp`. This function is documented on page 147.)

`\c_undefined_fp` A marker for undefined values.

```

7148 \tl_new:N \c_undefined_fp
7149 \tl_set:Nn \c_undefined_fp { X 0.000000000 e 0 }

```

(End definition for `\c_undefined_fp`. This function is documented on page 147.)

`\c_zero_fp` The constant zero value.

```

7150 \tl_new:N \c_zero_fp
7151 \tl_set:Nn \c_zero_fp { + 0.000000000 e 0 }

```

(End definition for `\c_zero_fp`. This function is documented on page 147.)

118.2 Variables

`\l_fp_arg_tl` A token list to store the formalised representation of the input for transcendental functions.

7152 `\tl_new:N \l_fp_arg_tl`

(End definition for \l_fp_arg_tl.)

`\l_fp_count_int` A counter for things like the number of divisions possible.

7153 `\int_new:N \l_fp_count_int`

(End definition for \l_fp_count_int.)

`\l_fp_div_offset_int` When carrying out division, an offset is used for the results to get the decimal part correct.

7154 `\int_new:N \l_fp_div_offset_int`

(End definition for \l_fp_div_offset_int.)

`\l_fp_exp_integer_int` Used for the calculation of exponent values.

`\l_fp_exp_decimal_int`

`\l_fp_exp_extended_int`

`\l_fp_exp_exponent_int`

7155 `\int_new:N \l_fp_exp_integer_int`

7156 `\int_new:N \l_fp_exp_decimal_int`

7157 `\int_new:N \l_fp_exp_extended_int`

7158 `\int_new:N \l_fp_exp_exponent_int`

(End definition for \l_fp_exp_integer_int.)

`\l_fp_input_a_sign_int` Storage for the input: two storage areas as there are at most two inputs.

`\l_fp_input_a_integer_int`

`\l_fp_input_a_decimal_int`

`\l_fp_input_a_exponent_int`

`\l_fp_input_b_sign_int`

`\l_fp_input_b_integer_int`

`\l_fp_input_b_decimal_int`

`\l_fp_input_b_exponent_int`

7159 `\int_new:N \l_fp_input_a_sign_int`

7160 `\int_new:N \l_fp_input_a_integer_int`

7161 `\int_new:N \l_fp_input_a_decimal_int`

7162 `\int_new:N \l_fp_input_a_exponent_int`

7163 `\int_new:N \l_fp_input_b_sign_int`

7164 `\int_new:N \l_fp_input_b_integer_int`

7165 `\int_new:N \l_fp_input_b_decimal_int`

7166 `\int_new:N \l_fp_input_b_exponent_int`

(End definition for \l_fp_input_a_sign_int.)

`\l_fp_input_a_extended_int` For internal use, ‘extended’ floating point numbers are needed.

`\l_fp_input_b_extended_int`

7167 `\int_new:N \l_fp_input_a_extended_int`

7168 `\int_new:N \l_fp_input_b_extended_int`

(End definition for \l_fp_input_a_extended_int.)

```

\l_fp_mul_a_i_int
\l_fp_mul_a_ii_int
\l_fp_mul_a_iii_int
\l_fp_mul_a_iv_int
\l_fp_mul_a_v_int
\l_fp_mul_a_vi_int
\l_fp_mul_b_i_int
\l_fp_mul_b_ii_int
\l_fp_mul_b_iii_int
\l_fp_mul_b_iv_int
\l_fp_mul_b_v_int
\l_fp_mul_b_vi_int

```

Multiplication requires that the decimal part is split into parts so that there are no overflows.

```

7169 \int_new:N \l_fp_mul_a_i_int
7170 \int_new:N \l_fp_mul_a_ii_int
7171 \int_new:N \l_fp_mul_a_iii_int
7172 \int_new:N \l_fp_mul_a_iv_int
7173 \int_new:N \l_fp_mul_a_v_int
7174 \int_new:N \l_fp_mul_a_vi_int
7175 \int_new:N \l_fp_mul_b_i_int
7176 \int_new:N \l_fp_mul_b_ii_int
7177 \int_new:N \l_fp_mul_b_iii_int
7178 \int_new:N \l_fp_mul_b_iv_int
7179 \int_new:N \l_fp_mul_b_v_int
7180 \int_new:N \l_fp_mul_b_vi_int

```

(End definition for `\l_fp_mul_a_i_int`.)

```

\l_fp_mul_output_int
\l_fp_mul_output_tl

```

Space for multiplication results.

```

7181 \int_new:N \l_fp_mul_output_int
7182 \tl_new:N \l_fp_mul_output_tl

```

(End definition for `\l_fp_mul_output_int`.)

```

\l_fp_output_sign_int
\l_fp_output_integer_int
\l_fp_output_decimal_int
\l_fp_output_exponent_int

```

Output is stored in the same way as input.

```

7183 \int_new:N \l_fp_output_sign_int
7184 \int_new:N \l_fp_output_integer_int
7185 \int_new:N \l_fp_output_decimal_int
7186 \int_new:N \l_fp_output_exponent_int

```

(End definition for `\l_fp_output_sign_int`.)

```

\l_fp_output_extended_int

```

Again, for calculations an extended part.

```

7187 \int_new:N \l_fp_output_extended_int

```

(End definition for `\l_fp_output_extended_int`.)

```

\l_fp_round_carry_bool

```

To indicate that a digit needs to be carried forward.

```

7188 \bool_new:N \l_fp_round_carry_bool

```

(End definition for `\l_fp_round_carry_bool`.)

```

\l_fp_round_decimal_tl

```

A temporary store when rounding, to build up the decimal part without needing to do any maths.

```

7189 \tl_new:N \l_fp_round_decimal_tl

```

(End definition for `\l_fp_round_decimal_tl`.)

`\l_fp_round_position_int` Used to check the position for rounding.
`\l_fp_round_target_int`

7190 `\int_new:N \l_fp_round_position_int`
7191 `\int_new:N \l_fp_round_target_int`

(End definition for `\l_fp_round_position_int`.)

`\l_fp_sign_tl` There are places where the sign needs to be set up ‘early’, so that the registers can be re-used.

7192 `\tl_new:N \l_fp_sign_tl`

(End definition for `\l_fp_sign_tl`.)

`\l_fp_split_sign_int` When splitting the input it is fastest to use a fixed name for the sign part, and to transfer it after the split is complete.

7193 `\int_new:N \l_fp_split_sign_int`

(End definition for `\l_fp_split_sign_int`.)

`\l_fp_tmp_int` A scratch `int`: used only where the value is not carried forward.

7194 `\int_new:N \l_fp_tmp_int`

(End definition for `\l_fp_tmp_int`.)

`\l_fp_tmp_tl` A scratch token list variable for expanding material.

7195 `\tl_new:N \l_fp_tmp_tl`

(End definition for `\l_fp_tmp_tl`.)

`\l_fp_trig_octant_int` To track which octant the trigonometric input is in.

7196 `\int_new:N \l_fp_trig_octant_int`

(End definition for `\l_fp_trig_octant_int`.)

`\l_fp_trig_sign_int` Used for the calculation of trigonometric values.
`\l_fp_trig_decimal_int`
`\l_fp_trig_extended_int`

7197 `\int_new:N \l_fp_trig_sign_int`
7198 `\int_new:N \l_fp_trig_decimal_int`
7199 `\int_new:N \l_fp_trig_extended_int`

(End definition for `\l_fp_trig_sign_int`.)

118.3 Parsing numbers

`\fp_read:N` Reading a stored value is made easier as the format is designed to match the delimited function. This is always used to read the first value (register `a`).

`\fp_read_aux:w`

```

7200 \cs_new_protected_nopar:Npn \fp_read:N #1 {
7201   \tex_expandafter:D \fp_read_aux:w #1 \q_stop
7202 }
7203 \cs_new_protected_nopar:Npn \fp_read_aux:w #1#2 . #3 e #4 \q_stop {
7204   \tex_if:D #1 -
7205     \l_fp_input_a_sign_int \c_minus_one
7206   \tex_else:D
7207     \l_fp_input_a_sign_int \c_one
7208   \tex_fi:D
7209   \l_fp_input_a_integer_int #2 \scan_stop:
7210   \l_fp_input_a_decimal_int #3 \scan_stop:
7211   \l_fp_input_a_exponent_int #4 \scan_stop:
7212 }

```

(End definition for `\fp_read:N`. This function is documented on page ??.)

`\fp_split:Nn`

`\fp_split_sign:`

`\fp_split_exponent:`

`\fp_split_aux_i:w`

`\fp_split_aux_ii:w`

`\fp_split_aux_iii:w`

`\fp_split_decimal:w`

`\fp_split_decimal_aux:w`

The aim here is to use as much of TeX's mechanism as possible to pick up the numerical input without any mistakes. In particular, negative numbers have to be filtered out first in case the integer part is 0 (in which case TeX would drop the `-` sign). That process has to be done in a loop for cases where the sign is repeated. Finding an exponent is relatively easy, after which the next phase is to find the integer part, which will terminate with a `.`, and trigger the decimal-finding code. The later will allow the decimal to be too long, truncating the result.

```

7213 \cs_new_protected_nopar:Npn \fp_split:Nn #1#2 {
7214   \tl_set:Nx \l_fp_tmp_tl {#2}
7215   \tl_set_rescan:Nno \l_fp_tmp_tl { \char_make_ignore:n { 32 } }
7216   { \l_fp_tmp_tl }
7217   \l_fp_split_sign_int \c_one
7218   \fp_split_sign:
7219   \use:c { l_fp_input_ #1 _sign_int } \l_fp_split_sign_int
7220   \tex_expandafter:D \fp_split_exponent:w \l_fp_tmp_tl e e \q_stop #1
7221 }
7222 \cs_new_protected_nopar:Npn \fp_split_sign: {
7223   \tex_ifnum:D \pdf_strcmp:D
7224     { \tex_expandafter:D \tl_head:w \l_fp_tmp_tl ? \q_stop } { - }
7225     = \c_zero
7226   \tl_set:Nx \l_fp_tmp_tl
7227     {
7228       \tex_expandafter:D
7229       \tl_tail:w \l_fp_tmp_tl \prg_do_nothing: \q_stop
7230     }
7231   \l_fp_split_sign_int -\l_fp_split_sign_int
7232   \tex_expandafter:D \fp_split_sign:

```



```

7233 \tex_else:D
7234 \tex_ifnum:D \pdf_strcmp:D
7235 { \tex_expandafter:D \tl_head:w \l_fp_tmp_tl ? \q_stop } { + }
7236 = \c_zero
7237 \tl_set:Nx \l_fp_tmp_tl
7238 {
7239 \tex_expandafter:D
7240 \tl_tail:w \l_fp_tmp_tl \prg_do_nothing: \q_stop
7241 }
7242 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7243 \fp_split_sign:
7244 \tex_fi:D
7245 \tex_fi:D
7246 }
7247 \cs_new_protected_nopar:Npn
7248 \fp_split_exponent:w #1 e #2 e #3 \q_stop #4 {
7249 \use:c { l_fp_input_ #4 _exponent_int }
7250 \etex_numexpr:D 0 #2 \scan_stop:
7251 \tex_afterassignment:D \fp_split_aux_i:w
7252 \use:c { l_fp_input_ #4 _integer_int }
7253 \etex_numexpr:D 0 #1 . . \q_stop #4
7254 }
7255 \cs_new_protected_nopar:Npn \fp_split_aux_i:w #1 . #2 . #3 \q_stop {
7256 \fp_split_aux_ii:w #2 000000000 \q_stop
7257 }
7258 \cs_new_protected_nopar:Npn \fp_split_aux_ii:w #1#2#3#4#5#6#7#8#9 {
7259 \fp_split_aux_iii:w {#1#2#3#4#5#6#7#8#9}
7260 }
7261 \cs_new_protected_nopar:Npn \fp_split_aux_iii:w #1#2 \q_stop {
7262 \l_fp_tmp_int 1 #1 \scan_stop:
7263 \tex_expandafter:D \fp_split_decimal:w
7264 \int_use:N \l_fp_tmp_int 000000000 \q_stop
7265 }
7266 \cs_new_protected_nopar:Npn \fp_split_decimal:w #1#2#3#4#5#6#7#8#9 {
7267 \fp_split_decimal_aux:w {#2#3#4#5#6#7#8#9}
7268 }
7269 \cs_new_protected_nopar:Npn \fp_split_decimal_aux:w #1#2#3 \q_stop #4 {
7270 \use:c { l_fp_input_ #4 _decimal_int } #1#2 \scan_stop:
7271 \tex_ifnum:D
7272 \etex_numexpr:D
7273 \use:c { l_fp_input_ #4 _integer_int } +
7274 \use:c { l_fp_input_ #4 _decimal_int }
7275 \scan_stop:
7276 = \c_zero
7277 \use:c { l_fp_input_ #4 _sign_int } \c_one
7278 \tex_fi:D
7279 \tex_ifnum:D
7280 \use:c { l_fp_input_ #4 _integer_int } < \c_one_thousand_million
7281 \tex_else:D
7282 \tex_expandafter:D \fp_overflow_msg:

```

```

7283 \tex_fi:D
7284 }

```

(End definition for `\fp_split:Nn`. This function is documented on page ??.)

`\fp_standardise:NNNN` The idea here is to shift the input into a known exponent range. This is done using \TeX tokens where possible, as this is faster than arithmetic.

```

\fp_standardise_aux:NNNN
\fp_standardise_aux:
\fp_standardise_aux:w
7285 \cs_new_protected_nopar:Npn \fp_standardise:NNNN #1#2#3#4 {
7286   \tex_ifnum:D
7287     \etex_numexpr:D #2 + #3 = \c_zero
7288     #1 \c_one
7289     #4 \c_zero
7290     \tex_expandafter:D \use_none:nnnn
7291   \tex_else:D
7292     \tex_expandafter:D \fp_standardise_aux:NNNN
7293   \tex_fi:D
7294   #1#2#3#4
7295 }
7296 \cs_new_protected_nopar:Npn \fp_standardise_aux:NNNN #1#2#3#4 {
7297   \cs_set_protected_nopar:Npn \fp_standardise_aux:
7298     {
7299       \tex_ifnum:D #2 = \c_zero
7300       \tex_advance:D #3 \c_one_thousand_million
7301       \tex_expandafter:D \fp_standardise_aux:w
7302       \int_use:N #3 \q_stop
7303       \tex_expandafter:D \fp_standardise_aux:
7304       \tex_fi:D
7305     }
7306   \cs_set_protected_nopar:Npn
7307     \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9 \q_stop
7308   {
7309     #2 ##2 \scan_stop:
7310     #3 ##3##4##5##6##7##8##9 0 \scan_stop:
7311     \tex_advance:D #4 \c_minus_one
7312   }
7313   \fp_standardise_aux:
7314   \cs_set_protected_nopar:Npn \fp_standardise_aux:
7315     {
7316       \tex_ifnum:D #2 > \c_nine
7317       \tex_advance:D #2 \c_one_thousand_million
7318       \tex_expandafter:D \use_i:nn \tex_expandafter:D
7319       \fp_standardise_aux:w \int_use:N #2
7320       \tex_expandafter:D \fp_standardise_aux:
7321       \tex_fi:D
7322     }
7323   \cs_set_protected_nopar:Npn
7324     \fp_standardise_aux:w ##1##2##3##4##5##6##7##8##9
7325   {
7326     #2 ##1##2##3##4##5##6##7##8 \scan_stop:

```

```

7327 \tex_advance:D #3 \c_one_thousand_million
7328 \tex_divide:D #3 \c_ten
7329 \tl_set:Nx \l_fp_tmp_tl
7330 {
7331   ##9
7332   \tex_expandafter:D \use_none:n \int_use:N #3
7333 }
7334 #3 \l_fp_tmp_tl \scan_stop:
7335 \tex_advance:D #4 \c_one
7336 }
7337 \fp_standardise_aux:
7338 \tex_ifnum:D #4 < \c_one_hundred
7339 \tex_ifnum:D #4 > -\c_one_hundred
7340 \tex_else:D
7341   #1 \c_one
7342   #2 \c_zero
7343   #3 \c_zero
7344   #4 \c_zero
7345 \tex_fi:D
7346 \tex_else:D
7347 \tex_expandafter:D \fp_overflow_msg:
7348 \tex_fi:D
7349 }
7350 \cs_new_protected_nopar:Npn \fp_standardise_aux: { }
7351 \cs_new_protected_nopar:Npn \fp_standardise_aux:w { }

```

(End definition for `\fp_standardise:NNNN`. This function is documented on page ??.)

118.4 Internal utilities

`\fp_level_input_exponents:` The routines here are similar to those used to standardise the exponent. However, the aim here is different: the two exponents need to end up the same.

```

\fp_level_input_exponents_a:NNNNNNNNN
\fp_level_input_exponents_b:NNNNNNNNN
7352 \cs_new_protected_nopar:Npn \fp_level_input_exponents: {
7353   \tex_ifnum:D \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
7354   \tex_expandafter:D \fp_level_input_exponents_a:
7355   \tex_else:D
7356   \tex_expandafter:D \fp_level_input_exponents_b:
7357   \tex_fi:D
7358 }
7359 \cs_new_protected_nopar:Npn \fp_level_input_exponents_a: {
7360   \tex_ifnum:D \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
7361   \tex_advance:D \l_fp_input_b_integer_int \c_one_thousand_million
7362   \tex_expandafter:D \use_i:nn \tex_expandafter:D
7363   \fp_level_input_exponents_a:NNNNNNNNN
7364   \int_use:N \l_fp_input_b_integer_int
7365   \tex_expandafter:D \fp_level_input_exponents_a:
7366   \tex_fi:D
7367 }

```

```

7368 \cs_new_protected_nopar:Npn
7369   \fp_level_input_exponents_a:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
7370   \l_fp_input_b_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
7371   \tex_advance:D \l_fp_input_b_decimal_int \c_one_thousand_million
7372   \tex_divide:D \l_fp_input_b_decimal_int \c_ten
7373   \tl_set:Nx \l_fp_tmp_tl
7374     {
7375       #9
7376       \tex_expandafter:D \use_none:n
7377       \int_use:N \l_fp_input_b_decimal_int
7378     }
7379   \l_fp_input_b_decimal_int \l_fp_tmp_tl \scan_stop:
7380   \tex_advance:D \l_fp_input_b_exponent_int \c_one
7381 }
7382 \cs_new_protected_nopar:Npn \fp_level_input_exponents_b: {
7383   \tex_ifnum:D \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
7384   \tex_advance:D \l_fp_input_a_integer_int \c_one_thousand_million
7385   \tex_expandafter:D \use_i:nn \tex_expandafter:D
7386   \fp_level_input_exponents_b:NNNNNNNNN
7387   \int_use:N \l_fp_input_a_integer_int
7388   \tex_expandafter:D \fp_level_input_exponents_b:
7389   \tex_fi:D
7390 }
7391 \cs_new_protected_nopar:Npn
7392   \fp_level_input_exponents_b:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
7393   \l_fp_input_a_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
7394   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7395   \tex_divide:D \l_fp_input_a_decimal_int \c_ten
7396   \tl_set:Nx \l_fp_tmp_tl
7397     {
7398       #9
7399       \tex_expandafter:D \use_none:n
7400       \int_use:N \l_fp_input_a_decimal_int
7401     }
7402   \l_fp_input_a_decimal_int \l_fp_tmp_tl \scan_stop:
7403   \tex_advance:D \l_fp_input_a_exponent_int \c_one
7404 }

```

(End definition for `\fp_level_input_exponents:`. This function is documented on page ??.)

`\fp_tmp:w` Used for output of results, cutting down on `\tex_expandafter:D`. This is just a placeholder definition.

```

7405 \cs_new_protected_nopar:Npn \fp_tmp:w #1#2 { }

```

(End definition for `\fp_tmp:w`.)

118.5 Operations for fp variables

The format of `fp` variables is tightly defined, so that they can be read quickly by the internal code. The format is a single sign token, a single number, the decimal point, nine decimal numbers, an `e` and finally the exponent. This final part may vary in length. When stored, floating points will always be stored with a value in the integer position unless the number is zero.

`\fp_new:N` Fixed-points always have a value, and of course this has to be initialised globally.

`\fp_new:c`

```

7406 \cs_new_protected_nopar:Npn \fp_new:N #1 {
7407   \tl_new:N #1
7408   \tl_gset_eq:NN #1 \c_zero_fp
7409 }
7410 \cs_generate_variant:Nn \fp_new:N { c }
```

(End definition for `\fp_new:N` and `\fp_new:c`. These functions are documented on page 148.)

`\fp_const:Nn` A simple wrapper.

`\fp_const:cn`

```

7411 \cs_new_protected_nopar:Npn \fp_const:Nn #1#2 {
7412   \cs_if_free:NTF #1
7413   {
7414     \fp_new:N #1
7415     \fp_gset:Nn #1 {#2}
7416   }
7417   {
7418     \msg_kernel_error:nx { variable-already-defined }
7419     { \token_to_str:N #1 }
7420   }
7421 }
7422 \cs_generate_variant:Nn \fp_const:Nn { c }
```

(End definition for `\fp_const:Nn` and `\fp_const:cn`. These functions are documented on page 148.)

`\fp_zero:N` Zeroing fixed-points is pretty obvious.

`\fp_zero:c`

`\fp_gzero:N`

`\fp_gzero:c`

```

7423 \cs_new_protected_nopar:Npn \fp_zero:N #1 {
7424   \tl_set_eq:NN #1 \c_zero_fp
7425 }
7426 \cs_new_protected_nopar:Npn \fp_gzero:N #1 {
7427   \tl_gset_eq:NN #1 \c_zero_fp
7428 }
7429 \cs_generate_variant:Nn \fp_zero:N { c }
7430 \cs_generate_variant:Nn \fp_gzero:N { c }
```

(End definition for `\fp_zero:N` and `\fp_zero:c`. These functions are documented on page 148.)

`\fp_set:Nn` To trap any input errors, a very simple version of the parser is run here. This will pick
`\fp_set:cn` up any invalid characters at this stage, saving issues later. The splitting approach is the
`\fp_gset:Nn` same as the more advanced function later.
`\fp_gset:cn`
`\fp_set_aux:NNn`

```

7431 \cs_new_protected_nopar:Npn \fp_set:Nn {
7432   \fp_set_aux:NNn \tl_set:Nn
7433 }
7434 \cs_new_protected_nopar:Npn \fp_gset:Nn {
7435   \fp_set_aux:NNn \tl_gset:Nn
7436 }
7437 \cs_new_protected_nopar:Npn \fp_set_aux:NNn #1#2#3 {
7438   \group_begin:
7439     \fp_split:Nn a {#3}
7440     \fp_standardise:NNNN
7441     \l_fp_input_a_sign_int
7442     \l_fp_input_a_integer_int
7443     \l_fp_input_a_decimal_int
7444     \l_fp_input_a_exponent_int
7445     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7446     \cs_set_protected_nopar:Npx \fp_tmp:w
7447     {
7448       \group_end:
7449       #1 \exp_not:N #2
7450       {
7451         \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7452         -
7453         \tex_else:D
7454         +
7455         \tex_fi:D
7456         \int_use:N \l_fp_input_a_integer_int
7457         .
7458         \tex_expandafter:D \use_none:n
7459         \int_use:N \l_fp_input_a_decimal_int
7460         e
7461         \int_use:N \l_fp_input_a_exponent_int
7462       }
7463     }
7464     \fp_tmp:w
7465   }
7466   \cs_generate_variant:Nn \fp_set:Nn { c }
7467   \cs_generate_variant:Nn \fp_gset:Nn { c }

```

(End definition for `\fp_set:Nn` and `\fp_set:cn`. These functions are documented on page 149.)

`\fp_set_from_dim:Nn` Here, dimensions are converted to fixed-points *via* a temporary variable. This ensures
`\fp_set_from_dim:cn` that they always convert as points. The code is then essentially the same as for `\fp_-`
`\fp_gset_from_dim:Nn` `set:Nn`, but with the dimension passed so that it will be striped of the `pt` on the way
`\fp_gset_from_dim:cn` through. The passage through a skip is used to remove any rubber part.
`\fp_set_from_dim_aux:NNn`
`\fp_set_from_dim_aux:w`

```

7468 \cs_new_protected_nopar:Npn \fp_set_from_dim:Nn {

```

423

```

7469 \fp_set_from_dim_aux:NNn \tl_set:Nx
7470 }
7471 \cs_new_protected_nopar:Npn \fp_gset_from_dim:Nn {
7472 \fp_set_from_dim_aux:NNn \tl_gset:Nx
7473 }
7474 \cs_new_protected_nopar:Npn \fp_set_from_dim_aux:NNn #1#2#3 {
7475 \group_begin:
7476 \l_fp_tmp_skip \etex_glueexpr:D #3 \scan_stop:
7477 \l_fp_tmp_dim \l_fp_tmp_skip
7478 \fp_split:Nn a
7479 {
7480 \tex_expandafter:D \fp_set_from_dim_aux:w
7481 \dim_use:N \l_fp_tmp_dim
7482 }
7483 \fp_standardise:NNNN
7484 \l_fp_input_a_sign_int
7485 \l_fp_input_a_integer_int
7486 \l_fp_input_a_decimal_int
7487 \l_fp_input_a_exponent_int
7488 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7489 \cs_set_protected_nopar:Npx \fp_tmp:w
7490 {
7491 \group_end:
7492 #1 \exp_not:N #2
7493 {
7494 \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7495 -
7496 \tex_else:D
7497 +
7498 \tex_fi:D
7499 \int_use:N \l_fp_input_a_integer_int
7500 .
7501 \tex_expandafter:D \use_none:n
7502 \int_use:N \l_fp_input_a_decimal_int
7503 e
7504 \int_use:N \l_fp_input_a_exponent_int
7505 }
7506 }
7507 \fp_tmp:w
7508 }
7509 \cs_set_protected_nopar:Npx \fp_set_from_dim_aux:w {
7510 \cs_set_nopar:Npn \exp_not:N \fp_set_from_dim_aux:w
7511 ##1 \tl_to_str:n { pt } {##1}
7512 }
7513 \fp_set_from_dim_aux:w
7514 \cs_generate_variant:Nn \fp_set_from_dim:Nn { c }
7515 \cs_generate_variant:Nn \fp_gset_from_dim:Nn { c }
7516 \dim_new:N \l_fp_tmp_dim
7517 \skip_new:N \l_fp_tmp_skip

```

(End definition for `\fp_set_from_dim:Nn` and `\fp_set_from_dim:cn`. These functions are documented on page 149.)

```

\fp_set_eq:NN Pretty simple, really.
\fp_set_eq:cN
\fp_set_eq:Nc
\fp_set_eq:cc
\fp_gset_eq:NN
\fp_gset_eq:cN
\fp_gset_eq:Nc
\fp_gset_eq:cc
7518 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
7519 \cs_new_eq:NN \fp_set_eq:cN \tl_set_eq:cN
7520 \cs_new_eq:NN \fp_set_eq:Nc \tl_set_eq:Nc
7521 \cs_new_eq:NN \fp_set_eq:cc \tl_set_eq:cc
7522 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
7523 \cs_new_eq:NN \fp_gset_eq:cN \tl_gset_eq:cN
7524 \cs_new_eq:NN \fp_gset_eq:Nc \tl_gset_eq:Nc
7525 \cs_new_eq:NN \fp_gset_eq:cc \tl_gset_eq:cc

```

(End definition for `\fp_set_eq:NN` and others. These functions are documented on page 148.)

```

\fp_show:N Simple showing of the underlying variable.
\fp_show:c

```

```

7526 \cs_new_eq:NN \fp_show:N \tl_show:N
7527 \cs_new_eq:NN \fp_show:c \tl_show:c

```

(End definition for `\fp_show:N` and `\fp_show:c`. These functions are documented on page 149.)

`\fp_use:N` The idea of the `\fp_use:N` function to convert the stored value into something suitable for \TeX to use as a number in an expandable manner. The first step is to deal with the sign, then work out how big the input is.

```

\fp_use_aux:w
\fp_use_none:w
\fp_use_small:w
\fp_use_large:w
\fp_use_large_aux_i:w
\fp_use_large_aux_1:w
\fp_use_large_aux_2:w
\fp_use_large_aux_3:w
\fp_use_large_aux_4:w
\fp_use_large_aux_5:w
\fp_use_large_aux_6:w
\fp_use_large_aux_7:w
\fp_use_large_aux_8:w
\fp_use_large_aux_i:w
\fp_use_large_aux_ii:w
7528 \cs_new_nopar:Npn \fp_use:N #1 {
7529   \tex_expandafter:D \fp_use_aux:w #1 \q_stop
7530 }
7531 \cs_generate_variant:Nn \fp_use:N { c }
7532 \cs_new_nopar:Npn \fp_use_aux:w #1#2 e #3 \q_stop {
7533   \tex_if:D #1 -
7534   -
7535   \tex_fi:D
7536   \tex_ifnum:D #3 > \c_zero
7537   \tex_expandafter:D \fp_use_large:w
7538   \tex_else:D
7539   \tex_ifnum:D #3 < \c_zero
7540   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7541   \fp_use_small:w
7542   \tex_else:D
7543   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7544   \fp_use_none:w
7545   \tex_fi:D
7546   \tex_fi:D
7547   #2 e #3 \q_stop
7548 }

```

When the exponent is zero, the input is simply returned as output.

```

7549 \cs_new_nopar:Npn \fp_use_none:w #1 e #2 \q_stop {#1}

```


For small numbers (less than 1) the correct number of zeros have to be inserted, but the decimal point is easy.

```

7550 \cs_new_nopar:Npn \fp_use_small:w #1 . #2 e #3 \q_stop {
7551   0 .
7552   \prg_replicate:nn { -#3 - 1 } { 0 }
7553   #1#2
7554 }

```

Life is more complex for large numbers. The decimal point needs to be shuffled, with potentially some zero-filling for very large values.

```

7555 \cs_new_nopar:Npn \fp_use_large:w #1 . #2 e #3 \q_stop {
7556   \tex_ifnum:D #3 < \c_ten
7557   \tex_expandafter:D \fp_use_large_aux_i:w
7558   \tex_else:D
7559   \tex_expandafter:D \fp_use_large_aux_ii:w
7560   \tex_fi:D
7561   #1#2 e #3 \q_stop
7562 }
7563 \cs_new_nopar:Npn \fp_use_large_aux_i:w #1#2 e #3 \q_stop {
7564   #1
7565   \use:c { fp_use_large_aux_ #3 :w } #2 \q_stop
7566 }
7567 \cs_new_nopar:cpn { fp_use_large_aux_1:w } #1#2 \q_stop { #1 . #2 }
7568 \cs_new_nopar:cpn { fp_use_large_aux_2:w } #1#2#3 \q_stop {
7569   #1#2 . #3
7570 }
7571 \cs_new_nopar:cpn { fp_use_large_aux_3:w } #1#2#3#4 \q_stop {
7572   #1#2#3 . #4
7573 }
7574 \cs_new_nopar:cpn { fp_use_large_aux_4:w } #1#2#3#4#5 \q_stop {
7575   #1#2#3#4 . #5
7576 }
7577 \cs_new_nopar:cpn { fp_use_large_aux_5:w } #1#2#3#4#5#6 \q_stop {
7578   #1#2#3#4#5 . #6
7579 }
7580 \cs_new_nopar:cpn { fp_use_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop {
7581   #1#2#3#4#5#6 . #7
7582 }
7583 \cs_new_nopar:cpn { fp_use_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop {
7584   #1#2#3#4#6#7 . #8
7585 }
7586 \cs_new_nopar:cpn { fp_use_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop {
7587   #1#2#3#4#5#6#7#8 . #9
7588 }
7589 \cs_new_nopar:cpn { fp_use_large_aux_9:w } #1 \q_stop { #1 . }
7590 \cs_new_nopar:Npn \fp_use_large_aux_ii:w #1 e #2 \q_stop {
7591   #1
7592   \prg_replicate:nn { #2 - 9 } { 0 }

```

```

7593 .
7594 }

```

(End definition for `\fp_use:N` and `\fp_use:c`. These functions are documented on page 149.)

118.6 Transferring to other types

The `\fp_use:N` function converts a floating point variable to a form that can be used by \TeX . Here, the functions are slightly different, as some information may be discarded.

`\fp_to_dim:N` A very simple wrapper.
`\fp_to_dim:c`

```

7595 \cs_new_nopar:Npn \fp_to_dim:N #1 { \fp_use:N #1 pt }
7596 \cs_generate_variant:Nn \fp_to_dim:N { c }

```

(End definition for `\fp_to_dim:N` and `\fp_to_dim:c`. These functions are documented on page 150.)

`\fp_to_int:N` Converting to integers in an expandable manner is very similar to simply using floating
`\fp_to_int:c` point variables, particularly in the lead-off.

```

\fp_to_int_aux:w
\fp_to_int_none:w
\fp_to_int_small:w
\fp_to_int_large:w
\fp_to_int_large_aux_i:w
\fp_to_int_large_aux_1:w
\fp_to_int_large_aux_2:w
\fp_to_int_large_aux_3:w
\fp_to_int_large_aux_4:w
\fp_to_int_large_aux_5:w
\fp_to_int_large_aux_6:w
\fp_to_int_large_aux_7:w
\fp_to_int_large_aux_8:w
\fp_to_int_large_aux_i:w
\fp_to_int_large_aux:nnn
\fp_to_int_large_aux_ii:w

```

```

7597 \cs_new_nopar:Npn \fp_to_int:N #1 {
7598   \tex_expandafter:D \fp_to_int_aux:w #1 \q_stop
7599 }
7600 \cs_generate_variant:Nn \fp_to_int:N { c }
7601 \cs_new_nopar:Npn \fp_to_int_aux:w #1#2 e #3 \q_stop {
7602   \tex_if:D #1 -
7603   -
7604   \tex_fi:D
7605   \tex_ifnum:D #3 < \c_zero
7606     \tex_expandafter:D \fp_to_int_small:w
7607   \tex_else:D
7608     \tex_expandafter:D \fp_to_int_large:w
7609   \tex_fi:D
7610   #2 e #3 \q_stop
7611 }

```

For small numbers, if the decimal part is greater than a half then there is rounding up to do.

```

7612 \cs_new_nopar:Npn \fp_to_int_small:w #1 . #2 e #3 \q_stop {
7613   \tex_ifnum:D #3 > \c_one
7614   \tex_else:D
7615     \tex_ifnum:D #1 < \c_five
7616     0
7617   \tex_else:D
7618     1
7619   \tex_fi:D
7620   \tex_fi:D
7621 }

```

For large numbers, the idea is to split off the part for rounding, do the rounding and fill if needed.

```

7622 \cs_new_nopar:Npn \fp_to_int_large:w #1 . #2 e #3 \q_stop {
7623   \tex_ifnum:D #3 < \c_ten
7624     \tex_expandafter:D \fp_to_int_large_aux_i:w
7625   \tex_else:D
7626     \tex_expandafter:D \fp_to_int_large_aux_ii:w
7627   \tex_fi:D
7628   #1#2 e #3 \q_stop
7629 }
7630 \cs_new_nopar:Npn \fp_to_int_large_aux_i:w #1#2 e #3 \q_stop {
7631   \use:c { fp_to_int_large_aux_ #3 :w } #2 \q_stop {#1}
7632 }
7633 \cs_new_nopar:cpn { fp_to_int_large_aux_1:w } #1#2 \q_stop {
7634   \fp_to_int_large_aux:nnn { #2 0 } {#1}
7635 }
7636 \cs_new_nopar:cpn { fp_to_int_large_aux_2:w } #1#2#3 \q_stop {
7637   \fp_to_int_large_aux:nnn { #3 00 } {#1#2}
7638 }
7639 \cs_new_nopar:cpn { fp_to_int_large_aux_3:w } #1#2#3#4 \q_stop {
7640   \fp_to_int_large_aux:nnn { #4 000 } {#1#2#3}
7641 }
7642 \cs_new_nopar:cpn { fp_to_int_large_aux_4:w } #1#2#3#4#5 \q_stop {
7643   \fp_to_int_large_aux:nnn { #5 0000 } {#1#2#3#4}
7644 }
7645 \cs_new_nopar:cpn { fp_to_int_large_aux_5:w } #1#2#3#4#5#6 \q_stop {
7646   \fp_to_int_large_aux:nnn { #6 00000 } {#1#2#3#4#5}
7647 }
7648 \cs_new_nopar:cpn { fp_to_int_large_aux_6:w } #1#2#3#4#5#6#7 \q_stop {
7649   \fp_to_int_large_aux:nnn { #7 000000 } {#1#2#3#4#5#6}
7650 }
7651 \cs_new_nopar:cpn
7652 { fp_to_int_large_aux_7:w } #1#2#3#4#5#6#7#8 \q_stop {
7653   \fp_to_int_large_aux:nnn { #8 0000000 } {#1#2#3#4#5#6#7}
7654 }
7655 \cs_new_nopar:cpn
7656 { fp_to_int_large_aux_8:w } #1#2#3#4#5#6#7#8#9 \q_stop {
7657   \fp_to_int_large_aux:nnn { #9 00000000 } {#1#2#3#4#5#6#7#8}
7658 }
7659 \cs_new_nopar:cpn { fp_to_int_large_aux_9:w } #1 \q_stop {#1}
7660 \cs_new_nopar:Npn \fp_to_int_large_aux:nnn #1#2#3 {
7661   \tex_ifnum:D #1 < \c_five_hundred_million
7662     #3#2
7663   \tex_else:D
7664     \tex_number:D \etex_numexpr:D #3#2 + 1 \scan_stop:
7665   \tex_fi:D
7666 }
7667 \cs_new_nopar:Npn \fp_to_int_large_aux_ii:w #1 e #2 \q_stop {
7668   #1

```

```

7669 \prg_replicate:nn { #2 - 9 } { 0 }
7670 }

```

(End definition for `\fp_to_int:N` and `\fp_to_int:c`. These functions are documented on page 150.)

`\fp_to_tl:N` Converting to integers in an expandable manner is very similar to simply using floating point variables, particularly in the lead-off.

`\fp_to_tl:c`

```

\fp_to_tl_aux:w
\fp_to_tl_large:w
\fp_to_tl_large_aux_i:w
\fp_to_tl_large_aux_ii:w
\fp_to_tl_large_0:w
\fp_to_tl_large_1:w
\fp_to_tl_large_2:w
\fp_to_tl_large_3:w
\fp_to_tl_large_4:w
\fp_to_tl_large_5:w
\fp_to_tl_large_6:w
\fp_to_tl_large_7:w
\fp_to_tl_large_8:w
\fp_to_tl_large_8_aux:w
\fp_to_tl_large_9:w
\fp_to_tl_small:w
\fp_to_tl_small_one:w
\fp_to_tl_small_two:w
\fp_to_tl_small_aux:w
\fp_to_tl_large_zeros:NNNNNNNNN
\fp_to_tl_small_zeros:NNNNNNNNN
\fp_use_iix_ix:NNNNNNNNN
\fp_use_ix:NNNNNNNNN
\fp_use_i_to_vii:NNNNNNNNN
\fp_use_i_to_iix:NNNNNNNNN
7671 \cs_new_nopar:Npn \fp_to_tl:N #1 {
7672   \tex_expandafter:D \fp_to_tl_aux:w #1 \q_stop
7673 }
7674 \cs_generate_variant:Nn \fp_to_tl:N { c }
7675 \cs_new_nopar:Npn \fp_to_tl_aux:w #1#2 e #3 \q_stop {
7676   \tex_if:D #1 -
7677   -
7678   \tex_fi:D
7679   \tex_ifnum:D #3 < \c_zero
7680     \tex_expandafter:D \fp_to_tl_small:w
7681   \tex_else:D
7682     \tex_expandafter:D \fp_to_tl_large:w
7683   \tex_fi:D
7684   #2 e #3 \q_stop
7685 }
For ‘large’ numbers (exponent  $\geq 0$ ) there are two cases. For very large exponents ( $\geq 10$ )
life is easy: apart from dropping extra zeros there is no work to do. On the other hand, for
intermediate exponent values the decimal needs to be moved, then zeros can be dropped.
7686 \cs_new_nopar:Npn \fp_to_tl_large:w #1 e #2 \q_stop {
7687   \tex_ifnum:D #2 < \c_ten
7688     \tex_expandafter:D \fp_to_tl_large_aux_i:w
7689   \tex_else:D
7690     \tex_expandafter:D \fp_to_tl_large_aux_ii:w
7691   \tex_fi:D
7692   #1 e #2 \q_stop
7693 }
7694 \cs_new_nopar:Npn \fp_to_tl_large_aux_i:w #1 e #2 \q_stop {
7695   \use:c { fp_to_tl_large_ #2 :w } #1 \q_stop
7696 }
7697 \cs_new_nopar:Npn \fp_to_tl_large_aux_ii:w #1 . #2 e #3 \q_stop {
7698   #1
7699   \fp_to_tl_large_zeros:NNNNNNNNN #2
7700   e #3
7701 }
7702 \cs_new_nopar:cpn { fp_to_tl_large_0:w } #1 . #2 \q_stop {
7703   #1
7704   \fp_to_tl_large_zeros:NNNNNNNNN #2
7705 }
7706 \cs_new_nopar:cpn { fp_to_tl_large_1:w } #1 . #2#3 \q_stop {
7707   #1#2

```

```

7708 \fp_to_tl_large_zeros:NNNNNNNN #3 0
7709 }
7710 \cs_new_nopar:cpn { fp_to_tl_large_2:w } #1 . #2#3#4 \q_stop {
7711 #1#2#3
7712 \fp_to_tl_large_zeros:NNNNNNNN #4 00
7713 }
7714 \cs_new_nopar:cpn { fp_to_tl_large_3:w } #1 . #2#3#4#5 \q_stop {
7715 #1#2#3#4
7716 \fp_to_tl_large_zeros:NNNNNNNN #5 000
7717 }
7718 \cs_new_nopar:cpn { fp_to_tl_large_4:w } #1 . #2#3#4#5#6 \q_stop {
7719 #1#2#3#4#5
7720 \fp_to_tl_large_zeros:NNNNNNNN #6 0000
7721 }
7722 \cs_new_nopar:cpn { fp_to_tl_large_5:w } #1 . #2#3#4#5#6#7 \q_stop {
7723 #1#2#3#4#5#6
7724 \fp_to_tl_large_zeros:NNNNNNNN #7 00000
7725 }
7726 \cs_new_nopar:cpn { fp_to_tl_large_6:w } #1 . #2#3#4#5#6#7#8 \q_stop {
7727 #1#2#3#4#5#6#7
7728 \fp_to_tl_large_zeros:NNNNNNNN #8 000000
7729 }
7730 \cs_new_nopar:cpn { fp_to_tl_large_7:w } #1 . #2#3#4#5#6#7#8#9 \q_stop {
7731 #1#2#3#4#5#6#7#8
7732 \fp_to_tl_large_zeros:NNNNNNNN #9 0000000
7733 }
7734 \cs_new_nopar:cpn { fp_to_tl_large_8:w } #1 . {
7735 #1
7736 \use:c { fp_to_tl_large_8_aux:w }
7737 }
7738 \cs_new_nopar:cpn
7739 { fp_to_tl_large_8_aux:w } #1#2#3#4#5#6#7#8#9 \q_stop {
7740 #1#2#3#4#5#6#7#8
7741 \fp_to_tl_large_zeros:NNNNNNNN #9 00000000
7742 }
7743 \cs_new_nopar:cpn { fp_to_tl_large_9:w } #1 . #2 \q_stop {#1#2}

```

Dealing with small numbers is a bit more complex as there has to be rounding. This makes life rather awkward, as there need to be a series of tests and calculations, as things cannot be stored in an expandable system.

```

7744 \cs_new_nopar:Npn \fp_to_tl_small:w #1 e #2 \q_stop {
7745 \tex_ifnum:D #2 = \c_minus_one
7746 \tex_expandafter:D \fp_to_tl_small_one:w
7747 \tex_else:D
7748 \tex_ifnum:D #2 = -\c_two
7749 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
7750 \fp_to_tl_small_two:w
7751 \tex_else:D
7752 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D

```

```

7753         \fp_to_tl_small_aux:w
7754     \tex_fi:D
7755     \tex_fi:D
7756     #1 e #2 \q_stop
7757 }
7758 \cs_new_nopar:Npn \fp_to_tl_small_one:w #1 . #2 e #3 \q_stop {
7759     \tex_ifnum:D \fp_use_ix:NNNNNNNNN #2 > \c_four
7760     \tex_ifnum:D
7761         \etex_numexpr:D #1 \fp_use_i_to_iix:NNNNNNNNN #2 + 1
7762         < \c_one_thousand_million
7763     0.
7764     \tex_expandafter:D \fp_to_tl_small_zeros:NNNNNNNNN
7765     \tex_number:D
7766         \etex_numexpr:D
7767         #1 \fp_use_i_to_iix:NNNNNNNNN #2 + 1
7768     \scan_stop:
7769     \tex_else:D
7770     1
7771     \tex_fi:D
7772     \tex_else:D
7773     0. #1
7774     \fp_to_tl_small_zeros:NNNNNNNNN #2
7775     \tex_fi:D
7776 }
7777 \cs_new_nopar:Npn \fp_to_tl_small_two:w #1 . #2 e #3 \q_stop {
7778     \tex_ifnum:D \fp_use_iix_ix:NNNNNNNNN #2 > \c_forty_four
7779     \tex_ifnum:D
7780     \etex_numexpr:D #1 \fp_use_i_to_vii:NNNNNNNNN #2 0 + \c_ten
7781     < \c_one_thousand_million
7782     0.0
7783     \tex_expandafter:D \fp_to_tl_small_zeros:NNNNNNNNN
7784     \tex_number:D
7785     \etex_numexpr:D
7786     #1 \fp_use_i_to_vii:NNNNNNNNN #2 0 + \c_ten
7787     \scan_stop:
7788     \tex_else:D
7789     0.1
7790     \tex_fi:D
7791     \tex_else:D
7792     0.0
7793     #1
7794     \fp_to_tl_small_zeros:NNNNNNNNN #2
7795     \tex_fi:D
7796 }
7797 \cs_new_nopar:Npn \fp_to_tl_small_aux:w #1 . #2 e #3 \q_stop {
7798     #1
7799     \fp_to_tl_large_zeros:NNNNNNNNN #2
7800     e #3
7801 }

```

Rather than a complex recursion, the tests for finding trailing zeros are written out long-hand. The difference between the two is only the need for a decimal marker.

```

7802 \cs_new_nopar:Npn \fp_to_tl_large_zeros:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
7803   \tex_ifnum:D #9 = \c_zero
7804     \tex_ifnum:D #8 = \c_zero
7805       \tex_ifnum:D #7 = \c_zero
7806         \tex_ifnum:D #6 = \c_zero
7807           \tex_ifnum:D #5 = \c_zero
7808             \tex_ifnum:D #4 = \c_zero
7809               \tex_ifnum:D #3 = \c_zero
7810                 \tex_ifnum:D #2 = \c_zero
7811                   \tex_ifnum:D #1 = \c_zero
7812                     \tex_else:D
7813                       . #1
7814                         \tex_fi:D
7815                       \tex_else:D
7816                         . #1#2
7817                           \tex_fi:D
7818                         \tex_else:D
7819                           . #1#2#3
7820                             \tex_fi:D
7821                           \tex_else:D
7822                             . #1#2#3#4
7823                               \tex_fi:D
7824                               \tex_else:D
7825                                 . #1#2#3#4#5
7826                                   \tex_fi:D
7827                                   \tex_else:D
7828                                     . #1#2#3#4#5#6
7829                                       \tex_fi:D
7830                                       \tex_else:D
7831                                         . #1#2#3#4#5#6#7
7832                                           \tex_fi:D
7833                                           \tex_else:D
7834                                             . #1#2#3#4#5#6#7#8
7835                                               \tex_fi:D
7836                                               \tex_else:D
7837                                                 . #1#2#3#4#5#6#7#8#9
7838                                                   \tex_fi:D
7839   }
7840 \cs_new_nopar:Npn \fp_to_tl_small_zeros:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
7841   \tex_ifnum:D #9 = \c_zero
7842     \tex_ifnum:D #8 = \c_zero
7843       \tex_ifnum:D #7 = \c_zero
7844         \tex_ifnum:D #6 = \c_zero
7845           \tex_ifnum:D #5 = \c_zero
7846             \tex_ifnum:D #4 = \c_zero
7847               \tex_ifnum:D #3 = \c_zero
7848                 \tex_ifnum:D #2 = \c_zero

```

```

7849         \tex_ifnum:D #1 = \c_zero
7850         \tex_else:D
7851             #1
7852         \tex_fi:D
7853     \tex_else:D
7854         #1#2
7855     \tex_fi:D
7856     \tex_else:D
7857         #1#2#3
7858     \tex_fi:D
7859     \tex_else:D
7860         #1#2#3#4
7861     \tex_fi:D
7862     \tex_else:D
7863         #1#2#3#4#5
7864     \tex_fi:D
7865     \tex_else:D
7866         #1#2#3#4#5#6
7867     \tex_fi:D
7868     \tex_else:D
7869         #1#2#3#4#5#6#7
7870     \tex_fi:D
7871     \tex_else:D
7872         #1#2#3#4#5#6#7#8
7873     \tex_fi:D
7874     \tex_else:D
7875         #1#2#3#4#5#6#7#8#9
7876     \tex_fi:D
7877 }

```

Some quick ‘return a few’ functions.

```

7878 \cs_new_nopar:Npn \fp_use_iix_ix:NNNNNNNN #1#2#3#4#5#6#7#8#9 {#8#9}
7879 \cs_new_nopar:Npn \fp_use_ix:NNNNNNNN #1#2#3#4#5#6#7#8#9 {#9}
7880 \cs_new_nopar:Npn \fp_use_i_to_vii:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
7881     #1#2#3#4#5#6#7
7882 }
7883 \cs_new_nopar:Npn \fp_use_i_to_iix:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
7884     #1#2#3#4#5#6#7#8
7885 }

```

(End definition for `\fp_to_tl:N` and `\fp_to_tl:c`. These functions are documented on page [150](#).)

118.7 Rounding numbers

The results may well need to be rounded. A couple of related functions to do this for a stored value.

`\fp_round_figures:Nn`
`\fp_round_figures:cn`
`\fp_ground_figures:Nn`
`\fp_ground_figures:cn`
`\fp_round_figures_aux:NNn`

Rounding to figures needs only an adjustment to the target by one (as the target is in decimal places).

```

7886 \cs_new_protected_nopar:Npn \fp_round_figures:Nn {
7887   \fp_round_figures_aux:NNn \tl_set:Nn
7888 }
7889 \cs_generate_variant:Nn \fp_round_figures:Nn { c }
7890 \cs_new_protected_nopar:Npn \fp_ground_figures:Nn {
7891   \fp_round_figures_aux:NNn \tl_gset:Nn
7892 }
7893 \cs_generate_variant:Nn \fp_ground_figures:Nn { c }
7894 \cs_new_protected_nopar:Npn \fp_round_figures_aux:NNn #1#2#3 {
7895   \group_begin:
7896     \fp_read:N #2
7897     \int_set:Nn \l_fp_round_target_int { #3 - 1 }
7898     \tex_ifnum:D \l_fp_round_target_int < \c_ten
7899       \tex_expandafter:D \fp_round:
7900     \tex_fi:D
7901     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7902     \cs_set_protected_nopar:Npx \fp_tmp:w
7903     {
7904       \group_end:
7905       #1 \exp_not:N #2
7906       {
7907         \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7908         -
7909         \tex_else:D
7910         +
7911         \tex_fi:D
7912         \int_use:N \l_fp_input_a_integer_int
7913         .
7914         \tex_expandafter:D \use_none:n
7915         \int_use:N \l_fp_input_a_decimal_int
7916         e
7917         \int_use:N \l_fp_input_a_exponent_int
7918       }
7919     }
7920     \fp_tmp:w
7921 }

```

(End definition for `\fp_round_figures:Nn` and `\fp_round_figures:cn`. These functions are documented on page 151.)

`\fp_round_places:Nn`
`\fp_round_places:cn`
`\fp_ground_places:Nn`
`\fp_ground_places:cn`
`\fp_round_places_aux:NNn`

Rounding to places needs an adjustment for the exponent value, which will mean that everything should be correct.

```

7922 \cs_new_protected_nopar:Npn \fp_round_places:Nn {
7923   \fp_round_places_aux:NNn \tl_set:Nn
7924 }
7925 \cs_generate_variant:Nn \fp_round_places:Nn { c }

```

```

7926 \cs_new_protected_nopar:Npn \fp_ground_places:Nn {
7927   \fp_round_places_aux:NNn \tl_gset:Nn
7928 }
7929 \cs_generate_variant:Nn \fp_ground_places:Nn { c }
7930 \cs_new_protected_nopar:Npn \fp_round_places_aux:NNn #1#2#3 {
7931   \group_begin:
7932     \fp_read:N #2
7933     \int_set:Nn \l_fp_round_target_int
7934       { #3 + \l_fp_input_a_exponent_int }
7935     \tex_ifnum:D \l_fp_round_target_int < \c_ten
7936     \tex_expandafter:D \fp_round:
7937     \tex_fi:D
7938     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7939     \cs_set_protected_nopar:Npx \fp_tmp:w
7940     {
7941       \group_end:
7942       #1 \exp_not:N #2
7943       {
7944         \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
7945         -
7946         \tex_else:D
7947         +
7948         \tex_fi:D
7949         \int_use:N \l_fp_input_a_integer_int
7950         .
7951         \tex_expandafter:D \use_none:n
7952         \int_use:N \l_fp_input_a_decimal_int
7953         e
7954         \int_use:N \l_fp_input_a_exponent_int
7955       }
7956     }
7957     \fp_tmp:w
7958 }

```

(End definition for `\fp_round_places:Nn` and `\fp_round_places:cn`. These functions are documented on page 151.)

`\fp_round:` The rounding approach is the same for decimal places and significant figures. There are always nine decimal digits to round, so the code can be written to account for this. The basic logic is simply to find the rounding, track any carry digit and move along. At the end of the loop there is a possible shuffle if the integer part has become 10.

```

7959 \cs_new_protected_nopar:Npn \fp_round: {
7960   \bool_set_false:N \l_fp_round_carry_bool
7961   \l_fp_round_position_int \c_eight
7962   \tl_clear:N \l_fp_round_decimal_tl
7963   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
7964   \tex_expandafter:D \use_i:nn \tex_expandafter:D
7965     \fp_round_aux:NNNNNNNNN \int_use:N \l_fp_input_a_decimal_int
7966 }

```

```

7967 \cs_new_protected_nopar:Npn \fp_round_aux:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
7968   \fp_round_loop:N #9#8#7#6#5#4#3#2#1
7969   \bool_if:NT \l_fp_round_carry_bool
7970     { \tex_advance:D \l_fp_input_a_integer_int \c_one }
7971   \l_fp_input_a_decimal_int \l_fp_round_decimal_tl \scan_stop:
7972   \tex_ifnum:D \l_fp_input_a_integer_int < \c_ten
7973   \tex_else:D
7974     \l_fp_input_a_integer_int \c_one
7975     \tex_divide:D \l_fp_input_a_decimal_int \c_ten
7976     \tex_advance:D \l_fp_input_a_exponent_int \c_one
7977   \tex_fi:D
7978 }
7979 \cs_new_protected_nopar:Npn \fp_round_loop:N #1 {
7980   \tex_ifnum:D \l_fp_round_position_int < \l_fp_round_target_int
7981     \bool_if:NTF \l_fp_round_carry_bool
7982       { \l_fp_tmp_int \etex_numexpr:D #1 + \c_one \scan_stop: }
7983       { \l_fp_tmp_int \etex_numexpr:D #1 \scan_stop: }
7984   \tex_ifnum:D \l_fp_tmp_int = \c_ten
7985     \l_fp_tmp_int \c_zero
7986   \tex_else:D
7987     \bool_set_false:N \l_fp_round_carry_bool
7988   \tex_fi:D
7989   \tl_set:Nx \l_fp_round_decimal_tl
7990     { \int_use:N \l_fp_tmp_int \l_fp_round_decimal_tl }
7991   \tex_else:D
7992     \tl_set:Nx \l_fp_round_decimal_tl { 0 \l_fp_round_decimal_tl }
7993   \tex_ifnum:D \l_fp_round_position_int = \l_fp_round_target_int
7994     \tex_ifnum:D #1 > \c_four
7995       \bool_set_true:N \l_fp_round_carry_bool
7996     \tex_fi:D
7997   \tex_fi:D
7998   \tex_fi:D
7999   \tex_advance:D \l_fp_round_position_int \c_minus_one
8000   \tex_ifnum:D \l_fp_round_position_int > \c_minus_one
8001     \tex_expandafter:D \fp_round_loop:N
8002   \tex_fi:D
8003 }

```

(End definition for `\fp_round:`. This function is documented on page ??.)

118.8 Unary functions

`\fp_abs:N` Setting the absolute value is easy: read the value, ignore the sign, return the result.

```

\fp_abs:c
\fp_gabs:N
\fp_gabs:c
\fp_abs_aux:NN
8004 \cs_new_protected_nopar:Npn \fp_abs:N {
8005   \fp_abs_aux:NN \tl_set:Nn
8006 }
8007 \cs_new_protected_nopar:Npn \fp_gabs:N {
8008   \fp_abs_aux:NN \tl_gset:Nn

```

```

8009 }
8010 \cs_generate_variant:Nn \fp_abs:N { c }
8011 \cs_generate_variant:Nn \fp_gabs:N { c }
8012 \cs_new_protected_nopar:Npn \fp_abs_aux:NN #1#2 {
8013   \group_begin:
8014     \fp_read:N #2
8015     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8016     \cs_set_protected_nopar:Npx \fp_tmp:w
8017     {
8018       \group_end:
8019       #1 \exp_not:N #2
8020       {
8021         +
8022         \int_use:N \l_fp_input_a_integer_int
8023         .
8024         \tex_expandafter:D \use_none:n
8025         \int_use:N \l_fp_input_a_decimal_int
8026         e
8027         \int_use:N \l_fp_input_a_exponent_int
8028       }
8029     }
8030   \fp_tmp:w
8031 }

```

(End definition for `\fp_abs:N` and `\fp_abs:c`. These functions are documented on page 152.)

`\fp_neg:N` Just a bit more complex: read the input, reverse the sign and output the result.

```

\fp_neg:c
\fp_gneg:N
\fp_gneg:c
\fp_neg:NN
8032 \cs_new_protected_nopar:Npn \fp_neg:N {
8033   \fp_neg_aux:NN \tl_set:Nn
8034 }
8035 \cs_new_protected_nopar:Npn \fp_gneg:N {
8036   \fp_neg_aux:NN \tl_gset:Nn
8037 }
8038 \cs_generate_variant:Nn \fp_neg:N { c }
8039 \cs_generate_variant:Nn \fp_gneg:N { c }
8040 \cs_new_protected_nopar:Npn \fp_neg_aux:NN #1#2 {
8041   \group_begin:
8042     \fp_read:N #2
8043     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8044     \tl_set:Nx \l_fp_tmp_tl
8045     {
8046       \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
8047       +
8048       \tex_else:D
8049       -
8050       \tex_fi:D
8051       \int_use:N \l_fp_input_a_integer_int
8052       .
8053       \tex_expandafter:D \use_none:n

```

```

8054         \int_use:N \l_fp_input_a_decimal_int
8055         e
8056         \int_use:N \l_fp_input_a_exponent_int
8057     }
8058 \tex_expandafter:D \group_end: \tex_expandafter:D
8059 #1 \tex_expandafter:D #2 \tex_expandafter:D { \l_fp_tmp_tl }
8060 }

```

(End definition for `\fp_neg:N` and `\fp_neg:c`. These functions are documented on page 152.)

118.9 Basic arithmetic

`\fp_add:Nn` The various addition functions are simply different ways to call the single master function below. This pattern is repeated for the other arithmetic functions.

```

8061 \cs_new_protected_nopar:Npn \fp_add:Nn {
8062   \fp_add_aux:NNn \tl_set:Nn
8063 }
8064 \cs_new_protected_nopar:Npn \fp_gadd:Nn {
8065   \fp_add_aux:NNn \tl_gset:Nn
8066 }
8067 \cs_generate_variant:Nn \fp_add:Nn { c }
8068 \cs_generate_variant:Nn \fp_gadd:Nn { c }

```

Addition takes place using one of two paths. If the signs of the two parts are the same, they are simply combined. On the other hand, if the signs are different the calculation finds this difference.

```

8069 \cs_new_protected_nopar:Npn \fp_add_aux:NNn #1#2#3 {
8070   \group_begin:
8071     \fp_read:N #2
8072     \fp_split:Nn b {#3}
8073     \fp_standardise:NNNN
8074     \l_fp_input_b_sign_int
8075     \l_fp_input_b_integer_int
8076     \l_fp_input_b_decimal_int
8077     \l_fp_input_b_exponent_int
8078     \fp_add_core:
8079     \fp_tmp:w #1#2
8080   }
8081 \cs_new_protected_nopar:Npn \fp_add_core: {
8082   \fp_level_input_exponents:
8083   \tex_ifnum:D
8084     \etex_numexpr:D
8085     \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8086     \scan_stop:
8087     > \c_zero
8088     \tex_expandafter:D \fp_add_sum:
8089   \tex_else:D

```

```

8090 \tex_expandafter:D \fp_add_difference:
8091 \tex_fi:D
8092 \l_fp_output_exponent_int \l_fp_input_a_exponent_int
8093 \fp_standardise:NNNN
8094 \l_fp_output_sign_int
8095 \l_fp_output_integer_int
8096 \l_fp_output_decimal_int
8097 \l_fp_output_exponent_int
8098 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8099 {
8100 \group_end:
8101 ##1 ##2
8102 {
8103 \tex_ifnum:D \l_fp_output_sign_int < \c_zero
8104 -
8105 \tex_else:D
8106 +
8107 \tex_fi:D
8108 \int_use:N \l_fp_output_integer_int
8109 .
8110 \tex_expandafter:D \use_none:n
8111 \tex_number:D \etex_numexpr:D
8112 \l_fp_output_decimal_int + \c_one_thousand_million
8113 e
8114 \int_use:N \l_fp_output_exponent_int
8115 }
8116 }
8117 }

```

Finding the sum of two numbers is trivially easy.

```

8118 \cs_new_protected_nopar:Npn \fp_add_sum: {
8119 \l_fp_output_sign_int \l_fp_input_a_sign_int
8120 \l_fp_output_integer_int
8121 \etex_numexpr:D
8122 \l_fp_input_a_integer_int + \l_fp_input_b_integer_int
8123 \scan_stop:
8124 \l_fp_output_decimal_int
8125 \etex_numexpr:D
8126 \l_fp_input_a_decimal_int + \l_fp_input_b_decimal_int
8127 \scan_stop:
8128 \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
8129 \tex_else:D
8130 \tex_advance:D \l_fp_output_integer_int \c_one
8131 \tex_advance:D \l_fp_output_decimal_int -\c_one_thousand_million
8132 \tex_fi:D
8133 }

```

When the signs of the two parts of the input are different, the absolute difference is worked out first. There is then a calculation to see which way around everything has

worked out, so that the final sign is correct. The difference might also give a zero result with a negative sign, which is reversed as zero is regarded as positive.

```

8134 \cs_new_protected_nopar:Npn \fp_add_difference: {
8135   \l_fp_output_integer_int
8136   \etex_numexpr:D
8137     \l_fp_input_a_integer_int - \l_fp_input_b_integer_int
8138   \scan_stop:
8139   \l_fp_output_decimal_int
8140   \etex_numexpr:D
8141     \l_fp_input_a_decimal_int - \l_fp_input_b_decimal_int
8142   \scan_stop:
8143   \tex_ifnum:D \l_fp_output_decimal_int < \c_zero
8144     \tex_advance:D \l_fp_output_integer_int \c_minus_one
8145     \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
8146   \tex_fi:D
8147   \tex_ifnum:D \l_fp_output_integer_int < \c_zero
8148     \l_fp_output_sign_int \l_fp_input_b_sign_int
8149     \tex_ifnum:D \l_fp_output_decimal_int = \c_zero
8150       \l_fp_output_integer_int - \l_fp_output_integer_int
8151     \tex_else:D
8152       \l_fp_output_decimal_int
8153       \etex_numexpr:D
8154         \c_one_thousand_million - \l_fp_output_decimal_int
8155       \scan_stop:
8156       \l_fp_output_integer_int
8157       \etex_numexpr:D
8158         - \l_fp_output_integer_int - \c_one
8159       \scan_stop:
8160     \tex_fi:D
8161   \tex_else:D
8162     \l_fp_output_sign_int \l_fp_input_a_sign_int
8163   \tex_fi:D
8164 }

```

(End definition for `\fp_add:Nn` and `\fp_add:cn`. These functions are documented on page 153.)

`\fp_sub:Nn` Subtraction is essentially the same as addition, but with the sign of the second component reversed. Thus the core of the two function groups is the same, with just a little set up here.

`\fp_sub:cn`

`\fp_gsub:Nn`

`\fp_gsub:cn`

`\fp_sub_aux:NNn`

```

8165 \cs_new_protected_nopar:Npn \fp_sub:Nn {
8166   \fp_sub_aux:NNn \tl_set:Nn
8167 }
8168 \cs_new_protected_nopar:Npn \fp_gsub:Nn {
8169   \fp_sub_aux:NNn \tl_gset:Nn
8170 }
8171 \cs_generate_variant:Nn \fp_sub:Nn { c }
8172 \cs_generate_variant:Nn \fp_gsub:Nn { c }
8173 \cs_new_protected_nopar:Npn \fp_sub_aux:NNn #1#2#3 {

```

```

8174 \group_begin:
8175   \fp_read:N #2
8176   \fp_split:Nn b {#3}
8177   \fp_standardise:NNNN
8178   \l_fp_input_b_sign_int
8179   \l_fp_input_b_integer_int
8180   \l_fp_input_b_decimal_int
8181   \l_fp_input_b_exponent_int
8182   \tex_multiply:D \l_fp_input_b_sign_int \c_minus_one
8183   \fp_add_core:
8184   \fp_tmp:w #1#2
8185 }

```

(End definition for `\fp_sub:Nn` and `\fp_sub:cn`. These functions are documented on page 153.)

```

\fp_mul:Nn The pattern is much the same for multiplication.
\fp_mul:cn
\fp_gmul:Nn
\fp_gmul:cn
\fp_mul_aux:NNn
\fp_mul_internal:
\fp_mul_split:NNNN
\fp_mul_split:w
\fp_mul_end_level:
\fp_mul_end_level:NNNNNNNN

```

```

8186 \cs_new_protected_nopar:Npn \fp_mul:Nn {
8187   \fp_mul_aux:NNn \tl_set:Nn
8188 }
8189 \cs_new_protected_nopar:Npn \fp_gmul:Nn {
8190   \fp_mul_aux:NNn \tl_gset:Nn
8191 }
8192 \cs_generate_variant:Nn \fp_mul:Nn { c }
8193 \cs_generate_variant:Nn \fp_gmul:Nn { c }

```

The approach to multiplication is as follows. First, the two numbers are split into blocks of three digits. These are then multiplied together to find products for each group of three output digits. This is all written out in full for speed reasons. Between each block of three digits in the output, there is a carry step. The very lowest digits are not calculated, while

```

8194 \cs_new_protected_nopar:Npn \fp_mul_aux:NNn #1#2#3 {
8195   \group_begin:
8196   \fp_read:N #2
8197   \fp_split:Nn b {#3}
8198   \fp_standardise:NNNN
8199   \l_fp_input_b_sign_int
8200   \l_fp_input_b_integer_int
8201   \l_fp_input_b_decimal_int
8202   \l_fp_input_b_exponent_int
8203   \fp_mul_internal:
8204   \l_fp_output_exponent_int
8205   \etex_numexpr:D
8206     \l_fp_input_a_exponent_int + \l_fp_input_b_exponent_int
8207   \scan_stop:
8208   \fp_standardise:NNNN
8209   \l_fp_output_sign_int
8210   \l_fp_output_integer_int
8211   \l_fp_output_decimal_int
8212   \l_fp_output_exponent_int

```



```

8213 \cs_set_protected_nopar:Npx \fp_tmp:w
8214 {
8215   \group_end:
8216   #1 \exp_not:N #2
8217   {
8218     \tex_ifnum:D
8219     \etex_numexpr:D
8220     \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8221     < \c_zero
8222     \tex_ifnum:D
8223     \etex_numexpr:D
8224     \l_fp_output_integer_int + \l_fp_output_decimal_int
8225     = \c_zero
8226     +
8227     \tex_else:D
8228     -
8229     \tex_fi:D
8230     \tex_else:D
8231     +
8232     \tex_fi:D
8233     \int_use:N \l_fp_output_integer_int
8234     .
8235     \tex_expandafter:D \use_none:n
8236     \tex_number:D \etex_numexpr:D
8237     \l_fp_output_decimal_int + \c_one_thousand_million
8238     e
8239     \int_use:N \l_fp_output_exponent_int
8240   }
8241 }
8242 \fp_tmp:w
8243 }

```

Done separately so that the internal use is a bit easier.

```

8244 \cs_new_protected_nopar:Npn \fp_mul_internal: {
8245   \fp_mul_split:NNNN \l_fp_input_a_decimal_int
8246   \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
8247   \fp_mul_split:NNNN \l_fp_input_b_decimal_int
8248   \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
8249   \l_fp_mul_output_int \c_zero
8250   \tl_clear:N \l_fp_mul_output_tl
8251   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int
8252   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
8253   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
8254   \tex_divide:D \l_fp_mul_output_int \c_one_thousand
8255   \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_iii_int
8256   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
8257   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
8258   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_input_b_integer_int
8259   \fp_mul_end_level:

```

```

8260 \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_ii_int
8261 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
8262 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_input_b_integer_int
8263 \fp_mul_end_level:
8264 \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_mul_b_i_int
8265 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_input_b_integer_int
8266 \fp_mul_end_level:
8267 \l_fp_output_decimal_int 0 \l_fp_mul_output_tl \scan_stop:
8268 \tl_clear:N \l_fp_mul_output_tl
8269 \fp_mul_product:NN \l_fp_input_a_integer_int \l_fp_input_b_integer_int
8270 \fp_mul_end_level:
8271 \l_fp_output_integer_int 0 \l_fp_mul_output_tl \scan_stop:
8272 }

```

The split works by making a 10 digit number, from which the first digit can then be dropped using a delimited argument. The groups of three digits are then assigned to the various parts of the input: notice that `##9` contains the last two digits of the smallest part of the input.

```

8273 \cs_new_protected_nopar:Npn \fp_mul_split:NNNN #1#2#3#4 {
8274 \tex_advance:D #1 \c_one_thousand_million
8275 \cs_set_protected_nopar:Npn \fp_mul_split_aux:w
8276   ##1##2##3##4##5##6##7##8##9 \q_stop {
8277   #2 ##2##3##4 \scan_stop:
8278   #3 ##5##6##7 \scan_stop:
8279   #4 ##8##9 \scan_stop:
8280 }
8281 \tex_expandafter:D \fp_mul_split_aux:w \int_use:N #1 \q_stop
8282 \tex_advance:D #1 -\c_one_thousand_million
8283 }
8284 \cs_new_protected_nopar:Npn \fp_mul_product:NN #1#2 {
8285 \l_fp_mul_output_int
8286 \etex_numexpr:D \l_fp_mul_output_int + #1 * #2 \scan_stop:
8287 }

```

At the end of each output group of three, there is a transfer of information so that there is no danger of an overflow. This is done by expansion to keep the number of calculations down.

```

8288 \cs_new_protected_nopar:Npn \fp_mul_end_level: {
8289 \tex_advance:D \l_fp_mul_output_int \c_one_thousand_million
8290 \tex_expandafter:D \use_i:nn \tex_expandafter:D
8291 \fp_mul_end_level:NNNNNNNNN \int_use:N \l_fp_mul_output_int
8292 }
8293 \cs_new_protected_nopar:Npn \fp_mul_end_level:NNNNNNNNN
8294   #1#2#3#4#5#6#7#8#9 {
8295 \tl_set:Nx \l_fp_mul_output_tl { #7#8#9 \l_fp_mul_output_tl }
8296 \l_fp_mul_output_int #1#2#3#4#5#6 \scan_stop:
8297 }

```

(End definition for `\fp_mul:Nn` and `\fp_mul:cn`. These functions are documented on page 153.)

`\fp_div:Nn` The pattern is much the same for multiplication.
`\fp_div:cn`
`\fp_gdiv:Nn`
`\fp_gdiv:cn`
`\fp_div_aux:NNn`
`\fp_div_internal:`
`\fp_div_loop:`
`\fp_div_divide:`
`\fp_div_divide_aux:`
`\fp_div_store:`
`\fp_div_store_integer:`
`\fp_div_store_decimal:`

```

8298 \cs_new_protected_nopar:Npn \fp_div:Nn {
8299   \fp_div_aux:NNn \tl_set:Nn
8300 }
8301 \cs_new_protected_nopar:Npn \fp_gdiv:Nn {
8302   \fp_div_aux:NNn \tl_gset:Nn
8303 }
8304 \cs_generate_variant:Nn \fp_div:Nn { c }
8305 \cs_generate_variant:Nn \fp_gdiv:Nn { c }

```

Division proper starts with a couple of tests. If the denominator is zero then a error is issued. On the other hand, if the numerator is zero then the result must be 0.0 and can be given with no further work.

```

8306 \cs_new_protected_nopar:Npn \fp_div_aux:NNn #1#2#3 {
8307   \group_begin:
8308     \fp_read:N #2
8309     \fp_split:Nn b {#3}
8310     \fp_standardise:NNNN
8311     \l_fp_input_b_sign_int
8312     \l_fp_input_b_integer_int
8313     \l_fp_input_b_decimal_int
8314     \l_fp_input_b_exponent_int
8315     \tex_ifnum:D
8316     \etex_numexpr:D
8317     \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
8318     = \c_zero
8319     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8320     {
8321       \group_end:
8322       #1 \exp_not:N #2 { \c_undefined_fp }
8323     }
8324     \tex_else:D
8325     \tex_ifnum:D
8326     \etex_numexpr:D
8327     \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
8328     = \c_zero
8329     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8330     {
8331       \group_end:
8332       #1 \exp_not:N #2 { \c_zero_fp }
8333     }
8334     \tex_else:D
8335     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8336     \fp_div_internal:
8337     \tex_fi:D
8338     \tex_fi:D
8339     \fp_tmp:w #1#2
8340 }

```

The main division algorithm works by finding how many times **b** can be removed from **a**, storing the result and doing the subtraction. Input **a** is then multiplied by 10, and the process is repeated. The looping ends either when there is nothing left of **a** (*i.e.* an exact result) or when the code reaches the ninth decimal place. Most of the process takes place in the loop function below.

```

8341 \cs_new_protected_nopar:Npn \fp_div_internal: {
8342   \l_fp_output_integer_int \c_zero
8343   \l_fp_output_decimal_int \c_zero
8344   \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
8345   \l_fp_div_offset_int \c_one_hundred_million
8346   \fp_div_loop:
8347   \l_fp_output_exponent_int
8348   \etex_numexpr:D
8349     \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int
8350   \scan_stop:
8351   \fp_standardise:NNNN
8352   \l_fp_output_sign_int
8353   \l_fp_output_integer_int
8354   \l_fp_output_decimal_int
8355   \l_fp_output_exponent_int
8356   \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
8357   {
8358     \group_end:
8359     ##1 ##2
8360     {
8361       \tex_ifnum:D
8362       \etex_numexpr:D
8363         \l_fp_input_a_sign_int * \l_fp_input_b_sign_int
8364       < \c_zero
8365       \tex_ifnum:D
8366       \etex_numexpr:D
8367         \l_fp_output_integer_int + \l_fp_output_decimal_int
8368       = \c_zero
8369       +
8370       \tex_else:D
8371       -
8372       \tex_fi:D
8373       \tex_else:D
8374       +
8375       \tex_fi:D
8376       \int_use:N \l_fp_output_integer_int
8377       .
8378       \tex_expandafter:D \use_none:n
8379       \tex_number:D \etex_numexpr:D
8380         \l_fp_output_decimal_int + \c_one_thousand_million
8381       \scan_stop:
8382     e
8383     \int_use:N \l_fp_output_exponent_int
8384   }

```

```

8385     }
8386 }

```

The main loop implements the approach described above. The storing function is done as a function so that the integer and decimal parts can be done separately but rapidly.

```

8387 \cs_new_protected_nopar:Npn \fp_div_loop: {
8388   \l_fp_count_int \c_zero
8389   \fp_div_divide:
8390   \fp_div_store:
8391   \tex_multiply:D \l_fp_input_a_integer_int \c_ten
8392   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8393   \tex_expandafter:D \fp_div_loop_step:w
8394     \int_use:N \l_fp_input_a_decimal_int \q_stop
8395   \tex_ifnum:D
8396     \etex_numexpr:D
8397       \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
8398       > \c_zero
8399     \tex_ifnum:D \l_fp_div_offset_int > \c_zero
8400       \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8401       \fp_div_loop:
8402     \tex_fi:D
8403   \tex_fi:D
8404 }

```

Checking to see if the numerator can be divided needs quite an involved check. Either the integer part has to be bigger for the numerator or, if it is not smaller then the decimal part of the numerator must not be smaller than that of the denominator. Once the test is right the rest is much as elsewhere.

```

8405 \cs_new_protected_nopar:Npn \fp_div_divide: {
8406   \tex_ifnum:D \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
8407   \tex_expandafter:D \fp_div_divide_aux:
8408   \tex_else:D
8409     \tex_ifnum:D \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
8410     \tex_else:D
8411       \tex_ifnum:D
8412         \l_fp_input_a_decimal_int < \l_fp_input_b_decimal_int
8413       \tex_else:D
8414         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8415         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8416         \tex_expandafter:D \fp_div_divide_aux:
8417       \tex_fi:D
8418     \tex_fi:D
8419   \tex_fi:D
8420 }
8421 \cs_new_protected_nopar:Npn \fp_div_divide_aux: {
8422   \tex_advance:D \l_fp_count_int \c_one
8423   \tex_advance:D \l_fp_input_a_integer_int -\l_fp_input_b_integer_int
8424   \tex_advance:D \l_fp_input_a_decimal_int -\l_fp_input_b_decimal_int

```

```

8425 \tex_ifnum:D \l_fp_input_a_decimal_int < \c_zero
8426 \tex_advance:D \l_fp_input_a_integer_int \c_minus_one
8427 \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8428 \tex_fi:D
8429 \fp_div_divide:
8430 }

```

Storing the number of each division is done differently for the integer and decimal. The integer is easy and a one-off, while the decimal also needs to account for the position of the digit to store.

```

8431 \cs_new_protected_nopar:Npn \fp_div_store: { }
8432 \cs_new_protected_nopar:Npn \fp_div_store_integer: {
8433   \l_fp_output_integer_int \l_fp_count_int
8434   \cs_set_eq:NN \fp_div_store: \fp_div_store_decimal:
8435 }
8436 \cs_new_protected_nopar:Npn \fp_div_store_decimal: {
8437   \l_fp_output_decimal_int
8438   \etex_numexpr:D
8439     \l_fp_output_decimal_int +
8440     \l_fp_count_int * \l_fp_div_offset_int
8441   \scan_stop:
8442   \tex_divide:D \l_fp_div_offset_int \c_ten
8443 }
8444 \cs_new_protected_nopar:Npn
8445   \fp_div_loop_step:w #1#2#3#4#5#6#7#8#9 \q_stop {
8446   \l_fp_input_a_integer_int
8447   \etex_numexpr:D
8448     #2 + \l_fp_input_a_integer_int
8449   \scan_stop:
8450   \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:
8451 }

```

(End definition for \fp_div:Nn and \fp_div:cn. These functions are documented on page ??.)

118.10 Arithmetic for internal use

For the more complex functions, it is only possible to deliver reliable 10 digit accuracy if the internal calculations are carried out to a higher degree of precision. This is done using a second set of functions so that the ‘user’ versions are not slowed down. These versions are also focussed on the needs of internal calculations. No error checking, sign checking or exponent levelling is done. For addition and subtraction, the arguments are:

- Integer part of input a.
- Decimal part of input a.
- Additional decimal part of input a.

- Integer part of input b.
- Decimal part of input b.
- Additional decimal part of input b.
- Integer part of output.
- Decimal part of output.
- Additional decimal part of output.

The situation for multiplication and division is a little different as they only deal with the decimal part.

`\fp_add:NNNNNNNN` The internal sum is always exactly that: it is always a sum and there is no sign check.

```

8452 \cs_new_protected_nopar:Npn \fp_add:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
8453   #7 \etex_numexpr:D #1 + #4 \scan_stop:
8454   #8 \etex_numexpr:D #2 + #5 \scan_stop:
8455   #9 \etex_numexpr:D #3 + #6 \scan_stop:
8456   \tex_ifnum:D #9 < \c_one_thousand_million
8457   \tex_else:D
8458     \tex_advance:D #8 \c_one
8459     \tex_advance:D #9 -\c_one_thousand_million
8460   \tex_fi:D
8461   \tex_ifnum:D #8 < \c_one_thousand_million
8462   \tex_else:D
8463     \tex_advance:D #7 \c_one
8464     \tex_advance:D #8 -\c_one_thousand_million
8465   \tex_fi:D
8466 }

```

(End definition for `\fp_add:NNNNNNNN`. This function is documented on page ??.)

`\fp_sub:NNNNNNNN` Internal subtraction is needed only when the first number is bigger than the second, so there is no need to worry about the sign. This is a good job as there are no arguments left. The flipping flag is used in the rare case where a sign change is possible.

```

8467 \cs_new_protected_nopar:Npn \fp_sub:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
8468   #7 \etex_numexpr:D #1 - #4 \scan_stop:
8469   #8 \etex_numexpr:D #2 - #5 \scan_stop:
8470   #9 \etex_numexpr:D #3 - #6 \scan_stop:
8471   \tex_ifnum:D #9 < \c_zero
8472     \tex_advance:D #8 \c_minus_one
8473     \tex_advance:D #9 \c_one_thousand_million
8474   \tex_fi:D
8475   \tex_ifnum:D #8 < \c_zero
8476     \tex_advance:D #7 \c_minus_one
8477     \tex_advance:D #8 \c_one_thousand_million

```

```

8478 \tex_fi:D
8479 \tex_ifnum:D #7 < \c_zero
8480 \tex_ifnum:D \etex_numexpr:D #8 + #9 = \c_zero
8481 #7 -#7
8482 \tex_else:D
8483 \tex_advance:D #7 \c_one
8484 #8 \etex_numexpr:D \c_one_thousand_million - #8 \scan_stop:
8485 #9 \etex_numexpr:D \c_one_thousand_million - #9 \scan_stop:
8486 \tex_fi:D
8487 \tex_fi:D
8488 }

```

(End definition for \fp_sub:NNNNNNNN. This function is documented on page ??.)

\fp_mul:NNNNNN Decimal-part only multiplication but with higher accuracy than the user version.

```

8489 \cs_new_protected_nopar:Npn \fp_mul:NNNNNN #1#2#3#4#5#6 {
8490 \fp_mul_split:NNNN #1
8491 \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
8492 \fp_mul_split:NNNN #2
8493 \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
8494 \fp_mul_split:NNNN #3
8495 \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
8496 \fp_mul_split:NNNN #4
8497 \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
8498 \l_fp_mul_output_int \c_zero
8499 \tl_clear:N \l_fp_mul_output_tl
8500 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_vi_int
8501 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_v_int
8502 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iv_int
8503 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_iii_int
8504 \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_ii_int
8505 \fp_mul_product:NN \l_fp_mul_a_vi_int \l_fp_mul_b_i_int
8506 \tex_divide:D \l_fp_mul_output_int \c_one_thousand
8507 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_v_int
8508 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iv_int
8509 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iii_int
8510 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_ii_int
8511 \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_i_int
8512 \fp_mul_end_level:
8513 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iv_int
8514 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iii_int
8515 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_ii_int
8516 \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_i_int
8517 \fp_mul_end_level:
8518 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int
8519 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
8520 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
8521 \fp_mul_end_level:
8522 #6 0 \l_fp_mul_output_tl \scan_stop:

```



```

8523 \tl_clear:N \l_fp_mul_output_tl
8524 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
8525 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
8526 \fp_mul_end_level:
8527 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
8528 \fp_mul_end_level:
8529 \fp_mul_end_level:
8530 #5 0 \l_fp_mul_output_tl \scan_stop:
8531 }

```

(End definition for `\fp_mul:NNNNNN`. This function is documented on page ??.)

`\fp_mul:NNNNNNNN` For internal multiplication where the integer does need to be retained. This means of course that this code is quite slow, and so is only used when necessary.

```

8532 \cs_new_protected_nopar:Npn \fp_mul:NNNNNNNN #1#2#3#4#5#6#7#8#9 {
8533   \fp_mul_split:NNNN #2
8534     \l_fp_mul_a_i_int \l_fp_mul_a_ii_int \l_fp_mul_a_iii_int
8535   \fp_mul_split:NNNN #3
8536     \l_fp_mul_a_iv_int \l_fp_mul_a_v_int \l_fp_mul_a_vi_int
8537   \fp_mul_split:NNNN #5
8538     \l_fp_mul_b_i_int \l_fp_mul_b_ii_int \l_fp_mul_b_iii_int
8539   \fp_mul_split:NNNN #6
8540     \l_fp_mul_b_iv_int \l_fp_mul_b_v_int \l_fp_mul_b_vi_int
8541   \l_fp_mul_output_int \c_zero
8542   \tl_clear:N \l_fp_mul_output_tl
8543   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_vi_int
8544   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_v_int
8545   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iv_int
8546   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_iii_int
8547   \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_ii_int
8548   \fp_mul_product:NN \l_fp_mul_a_vi_int \l_fp_mul_b_i_int
8549   \tex_divide:D \l_fp_mul_output_int \c_one_thousand
8550   \fp_mul_product:NN #1 \l_fp_mul_b_vi_int
8551   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_v_int
8552   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iv_int
8553   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_iii_int
8554   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_ii_int
8555   \fp_mul_product:NN \l_fp_mul_a_v_int \l_fp_mul_b_i_int
8556   \fp_mul_product:NN \l_fp_mul_a_vi_int #4
8557   \fp_mul_end_level:
8558   \fp_mul_product:NN #1 \l_fp_mul_b_v_int
8559   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iv_int
8560   \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_iii_int
8561   \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_ii_int
8562   \fp_mul_product:NN \l_fp_mul_a_iv_int \l_fp_mul_b_i_int
8563   \fp_mul_product:NN \l_fp_mul_a_v_int #4
8564   \fp_mul_end_level:
8565   \fp_mul_product:NN #1 \l_fp_mul_b_iv_int
8566   \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_iii_int

```

```

8567 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_ii_int
8568 \fp_mul_product:NN \l_fp_mul_a_iii_int \l_fp_mul_b_i_int
8569 \fp_mul_product:NN \l_fp_mul_a_iv_int #4
8570 \fp_mul_end_level:
8571 #9 0 \l_fp_mul_output_tl \scan_stop:
8572 \tl_clear:N \l_fp_mul_output_tl
8573 \fp_mul_product:NN #1 \l_fp_mul_b_iii_int
8574 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_ii_int
8575 \fp_mul_product:NN \l_fp_mul_a_ii_int \l_fp_mul_b_i_int
8576 \fp_mul_product:NN \l_fp_mul_a_iii_int #4
8577 \fp_mul_end_level:
8578 \fp_mul_product:NN #1 \l_fp_mul_b_ii_int
8579 \fp_mul_product:NN \l_fp_mul_a_i_int \l_fp_mul_b_i_int
8580 \fp_mul_product:NN \l_fp_mul_a_ii_int #4
8581 \fp_mul_end_level:
8582 \fp_mul_product:NN #1 \l_fp_mul_b_i_int
8583 \fp_mul_product:NN \l_fp_mul_a_i_int #4
8584 \fp_mul_end_level:
8585 #8 0 \l_fp_mul_output_tl \scan_stop:
8586 \tl_clear:N \l_fp_mul_output_tl
8587 \fp_mul_product:NN #1 #4
8588 \fp_mul_end_level:
8589 #7 0 \l_fp_mul_output_tl \scan_stop:
8590 }

```

(End definition for `\fp_mul:NNNNNNNN`. This function is documented on page ??.)

`\fp_div_integer:NNNNN` Here, division is always by an integer, and so it is possible to use TeX's native calculations rather than doing it in macros. The idea here is to divide the decimal part, find any remainder, then do the real division of the two parts before adding in what is needed for the remainder.

```

8591 \cs_new_protected_nopar:Npn \fp_div_integer:NNNNN #1#2#3#4#5 {
8592   \l_fp_tmp_int #1
8593   \tex_divide:D \l_fp_tmp_int #3
8594   \l_fp_tmp_int \etex_numexpr:D #1 - \l_fp_tmp_int * #3 \scan_stop:
8595   #4 #1
8596   \tex_divide:D #4 #3
8597   #5 #2
8598   \tex_divide:D #5 #3
8599   \tex_multiply:D \l_fp_tmp_int \c_one_thousand
8600   \tex_divide:D \l_fp_tmp_int #3
8601   #5 \etex_numexpr:D #5 + \l_fp_tmp_int * \c_one_million \scan_stop:
8602   \tex_ifnum:D #5 > \c_one_thousand_million
8603     \tex_advance:D #4 \c_one
8604     \tex_advancd:D #5 -\c_one_thousand_million
8605   \tex_fi:D
8606 }

```

(End definition for `\fp_div_integer:NNNNN`. This function is documented on page ??.)

\fp_extended_normalise: The ‘extended’ integers for internal use are mainly used in fixed-point mode. This comes up in a few places, so a generalised utility is made available to carry out the change. This function simply calls the two loops to shift the input to the point of having a zero exponent.

```

\fp_extended_normalise_aux_i:
\fp_extended_normalise_aux_i:w
\fp_extended_normalise_aux_ii:w
\fp_extended_normalise_aux_ii:
\fp_extended_normalise_aux:NNNNNNNNN
8607 \cs_new_protected_nopar:Npn \fp_extended_normalise: {
8608   \fp_extended_normalise_aux_i:
8609   \fp_extended_normalise_aux_ii:
8610 }
8611 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_i: {
8612   \tex_ifnum:D \l_fp_input_a_exponent_int > \c_zero
8613   \tex_multiply:D \l_fp_input_a_integer_int \c_ten
8614   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8615   \tex_expandafter:D \fp_extended_normalise_aux_i:w
8616   \int_use:N \l_fp_input_a_decimal_int \q_stop
8617   \tex_expandafter:D \fp_extended_normalise_aux_i:
8618   \tex_fi:D
8619 }
8620 \cs_new_protected_nopar:Npn
8621   \fp_extended_normalise_aux_i:w #1#2#3#4#5#6#7#8#9 \q_stop {
8622   \l_fp_input_a_integer_int
8623   \etex_numexpr:D \l_fp_input_a_integer_int + #2 \scan_stop:
8624   \l_fp_input_a_decimal_int #3#4#5#6#7#8#9 0 \scan_stop:
8625   \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
8626   \tex_expandafter:D \fp_extended_normalise_aux_ii:w
8627   \int_use:N \l_fp_input_a_extended_int \q_stop
8628 }
8629 \cs_new_protected_nopar:Npn
8630   \fp_extended_normalise_aux_ii:w #1#2#3#4#5#6#7#8#9 \q_stop {
8631   \l_fp_input_a_decimal_int
8632   \etex_numexpr:D \l_fp_input_a_decimal_int + #2 \scan_stop:
8633   \l_fp_input_a_extended_int #3#4#5#6#7#8#9 0 \scan_stop:
8634   \tex_advance:D \l_fp_input_a_exponent_int \c_minus_one
8635 }
8636 \cs_new_protected_nopar:Npn \fp_extended_normalise_aux_ii: {
8637   \tex_ifnum:D \l_fp_input_a_exponent_int < \c_zero
8638   \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8639   \tex_expandafter:D \use_i:nn \tex_expandafter:D
8640     \fp_extended_normalise_ii_aux:NNNNNNNNN
8641   \int_use:N \l_fp_input_a_decimal_int
8642   \tex_expandafter:D \fp_extended_normalise_aux_ii:
8643   \tex_fi:D
8644 }
8645 \cs_new_protected_nopar:Npn
8646   \fp_extended_normalise_ii_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
8647   \tex_ifnum:D \l_fp_input_a_integer_int = \c_zero
8648   \l_fp_input_a_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
8649   \tex_else:D
8650     \tl_set:Nx \l_fp_tmp_tl
8651     {

```

```

8652         \int_use:N \l_fp_input_a_integer_int
8653         #1#2#3#4#5#6#7#8
8654     }
8655     \l_fp_input_a_integer_int \c_zero
8656     \l_fp_input_a_decimal_int \l_fp_tmp_tl \scan_stop:
8657 \tex_fi:D
8658 \tex_divide:D \l_fp_input_a_extended_int \c_ten
8659 \tl_set:Nx \l_fp_tmp_tl
8660 {
8661     #9
8662     \int_use:N \l_fp_input_a_extended_int
8663 }
8664 \l_fp_input_a_extended_int \l_fp_tmp_tl \scan_stop:
8665 \tex_advance:D \l_fp_input_a_exponent_int \c_one
8666 }

```

(End definition for `\fp_extended_normalise:.` This function is documented on page ??.)

`\fp_extended_normalise_output:` At some stages in working out extended output, it is possible for the value to need shifting to keep the integer part in range. This only ever happens such that the integer needs to be made smaller.

`\fp_extended_normalise_output_aux_i:NNNNNNNNN`
`\fp_extended_normalise_output_aux_ii:NNNNNNNNN`
`\fp_extended_normalise_output_aux:N`

```

8667 \cs_new_protected_nopar:Npn \fp_extended_normalise_output: {
8668     \tex_ifnum:D \l_fp_output_integer_int > \c_nine
8669     \tex_advance:D \l_fp_output_integer_int \c_one_thousand_million
8670     \tex_expandafter:D \use_i:nn \tex_expandafter:D
8671         \fp_extended_normalise_output_aux_i:NNNNNNNNN
8672         \int_use:N \l_fp_output_integer_int
8673     \tex_expandafter:D \fp_extended_normalise_output:
8674     \tex_fi:D
8675 }
8676 \cs_new_protected_nopar:Npn
8677     \fp_extended_normalise_output_aux_i:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
8678     \l_fp_output_integer_int #1#2#3#4#5#6#7#8 \scan_stop:
8679     \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
8680     \tl_set:Nx \l_fp_tmp_tl
8681     {
8682         #9
8683         \tex_expandafter:D \use_none:n
8684         \int_use:N \l_fp_output_decimal_int
8685     }
8686     \tex_expandafter:D \fp_extended_normalise_output_aux_ii:NNNNNNNNN
8687     \l_fp_tmp_tl
8688 }
8689 \cs_new_protected_nopar:Npn
8690     \fp_extended_normalise_output_aux_ii:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
8691     \l_fp_output_decimal_int #1#2#3#4#5#6#7#8#9 \scan_stop:
8692     \fp_extended_normalise_output_aux:N
8693 }
8694 \cs_new_protected_nopar:Npn \fp_extended_normalise_output_aux:N #1 {

```

```

8695 \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
8696 \tex_divide:D \l_fp_output_extended_int \c_ten
8697 \tl_set:Nx \l_fp_tmp_tl
8698 {
8699   #1
8700   \tex_expandafter:D \use_none:n
8701   \int_use:N \l_fp_output_extended_int
8702 }
8703 \l_fp_output_extended_int \l_fp_tmp_tl \scan_stop:
8704 \tex_advance:D \l_fp_output_exponent_int \c_one
8705 }

```

(End definition for `\fp_extended_normalise_output:`. This function is documented on page ??.)

118.11 Trigonometric functions

`\fp_trig_normalise:` For normalisation, the code essentially switches to fixed-point arithmetic. There is a shift of the exponent, then repeated subtractions. The end result is a number in the range $-\pi < x \leq \pi$.

`\fp_trig_normalise_aux:`

`\fp_trig_sub:NNN`

```

8706 \cs_new_protected_nopar:Npn \fp_trig_normalise: {
8707   \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
8708   \l_fp_input_a_extended_int \c_zero
8709   \fp_extended_normalise:
8710   \fp_trig_normalise_aux:
8711   \tex_ifnum:D \l_fp_input_a_integer_int < \c_zero
8712   \l_fp_input_a_sign_int -\l_fp_input_a_sign_int
8713   \l_fp_input_a_integer_int -\l_fp_input_a_integer_int
8714   \tex_fi:D
8715   \tex_expandafter:D \fp_trig_octant:
8716 \tex_else:D
8717   \l_fp_input_a_sign_int \c_one
8718   \l_fp_output_integer_int \c_zero
8719   \l_fp_output_decimal_int \c_zero
8720   \l_fp_output_exponent_int \c_zero
8721   \tex_expandafter:D \fp_trig_overflow_msg:
8722   \tex_fi:D
8723 }
8724 \cs_new_protected_nopar:Npn \fp_trig_normalise_aux: {
8725   \tex_ifnum:D \l_fp_input_a_integer_int > \c_three
8726   \fp_trig_sub:NNN
8727   \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
8728   \tex_expandafter:D \fp_trig_normalise_aux:
8729 \tex_else:D
8730   \tex_ifnum:D \l_fp_input_a_integer_int > \c_two
8731   \tex_ifnum:D \l_fp_input_a_decimal_int > \c_fp_pi_decimal_int
8732   \fp_trig_sub:NNN
8733   \c_six \c_fp_two_pi_decimal_int \c_fp_two_pi_extended_int
8734   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D

```

```

8735         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8736         \tex_expandafter:D \fp_trig_normalise_aux:
8737         \tex_fi:D
8738     \tex_fi:D
8739 \tex_fi:D
8740 }

```

Here, there may be a sign change but there will never be any variation in the input. So a dedicated function can be used.

```

8741 \cs_new_protected_nopar:Npn \fp_trig_sub:NNN #1#2#3 {
8742     \l_fp_input_a_integer_int
8743     \etex_numexpr:D \l_fp_input_a_integer_int - #1 \scan_stop:
8744     \l_fp_input_a_decimal_int
8745     \etex_numexpr:D \l_fp_input_a_decimal_int - #2 \scan_stop:
8746     \l_fp_input_a_extended_int
8747     \etex_numexpr:D \l_fp_input_a_extended_int - #3 \scan_stop:
8748     \tex_ifnum:D \l_fp_input_a_extended_int < \c_zero
8749     \tex_advance:D \l_fp_input_a_decimal_int \c_minus_one
8750     \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
8751     \tex_fi:D
8752     \tex_ifnum:D \l_fp_input_a_decimal_int < \c_zero
8753     \tex_advance:D \l_fp_input_a_integer_int \c_minus_one
8754     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million
8755     \tex_fi:D
8756     \tex_ifnum:D \l_fp_input_a_integer_int < \c_zero
8757     \l_fp_input_a_sign_int -\l_fp_input_a_sign_int
8758     \tex_ifnum:D
8759     \etex_numexpr:D
8760         \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
8761     = \c_zero
8762     \l_fp_input_a_integer_int -\l_fp_input_a_integer_int
8763     \tex_else:D
8764     \l_fp_input_a_integer_int
8765     \etex_numexpr:D
8766         - \l_fp_input_a_integer_int - \c_one
8767     \scan_stop:
8768     \l_fp_input_a_decimal_int
8769     \etex_numexpr:D
8770         \c_one_thousand_million - \l_fp_input_a_decimal_int
8771     \scan_stop:
8772     \l_fp_input_a_extended_int
8773     \etex_numexpr:D
8774         \c_one_thousand_million - \l_fp_input_a_extended_int
8775     \scan_stop:
8776     \tex_fi:D
8777 \tex_fi:D
8778 }

```

(End definition for `\fp_trig_normalise:.` This function is documented on page ??.)

`\fp_trig_octant:` Here, the input is further reduced into the range $0 \leq x < \pi/4$. This is pretty simple:
`\fp_trig_octant_aux:` check if $\pi/4$ can be taken off and if it can do it and loop. The check at the end is to ‘mop up’ values which are so close to $\pi/4$ that they should be treated as such. The test for an even octant is needed as the ‘remainder’ needed is from the nearest $\pi/2$.

```

8779 \cs_new_protected_nopar:Npn \fp_trig_octant: {
8780   \l_fp_trig_octant_int \c_one
8781   \fp_trig_octant_aux:
8782   \tex_ifnum:D \l_fp_input_a_decimal_int < \c_ten
8783     \l_fp_input_a_decimal_int \c_zero
8784     \l_fp_input_a_extended_int \c_zero
8785   \tex_fi:D
8786   \tex_ifodd:D \l_fp_trig_octant_int
8787   \tex_else:D
8788     \fp_sub:NNNNNNNNN
8789     \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
8790     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8791     \l_fp_input_a_extended_int
8792     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8793     \l_fp_input_a_extended_int
8794   \tex_fi:D
8795 }
8796 \cs_new_protected_nopar:Npn \fp_trig_octant_aux: {
8797   \tex_ifnum:D \l_fp_input_a_integer_int > \c_zero
8798   \fp_sub:NNNNNNNNN
8799   \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8800   \l_fp_input_a_extended_int
8801   \c_zero \c_fp_pi_by_four_decimal_int \c_fp_pi_by_four_extended_int
8802   \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8803   \l_fp_input_a_extended_int
8804   \tex_advance:D \l_fp_trig_octant_int \c_one
8805   \tex_expandafter:D \fp_trig_octant_aux:
8806   \tex_else:D
8807   \tex_ifnum:D
8808     \l_fp_input_a_decimal_int > \c_fp_pi_by_four_decimal_int
8809     \fp_sub:NNNNNNNNN
8810     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8811     \l_fp_input_a_extended_int
8812     \c_zero \c_fp_pi_by_four_decimal_int
8813     \c_fp_pi_by_four_extended_int
8814     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
8815     \l_fp_input_a_extended_int
8816     \tex_advance:D \l_fp_trig_octant_int \c_one
8817     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8818     \fp_trig_octant_aux:
8819     \tex_fi:D
8820   \tex_fi:D
8821 }

```

(End definition for `\fp_trig_octant:`. This function is documented on page ??.)

`\fp_sin:Nn` Calculating the sine starts off in the usual way. There is a check to see if the value has
`\fp_sin:cn` already been worked out before proceeding further.
`\fp_gsin:Nn`
`\fp_gsin:cn`

```

8822 \cs_new_protected_nopar:Npn \fp_sin:Nn {
8823   \fp_sin_aux:NNn \tl_set:Nn
8824 }
\fp_sin_aux:NNn
8825 \cs_new_protected_nopar:Npn \fp_gsin:Nn {
\fp_sin_aux_i:
8826   \fp_sin_aux:NNn \tl_gset:Nn
8827 }
8828 \cs_generate_variant:Nn \fp_sin:Nn { c }
\fp_sin_aux_ii:
8829 \cs_generate_variant:Nn \fp_gsin:Nn { c }

```

The internal routine for sines does a check to see if the value is already known. This saves a lot of repetition when doing rotations. For very small values it is best to simply return the input as the sine: the cut-off is 1×10^{-5} .

```

8830 \cs_new_protected_nopar:Npn \fp_sin_aux:NNn #1#2#3 {
8831   \group_begin:
8832     \fp_split:Nn a {#3}
8833     \fp_standardise:NNNN
8834     \l_fp_input_a_sign_int
8835     \l_fp_input_a_integer_int
8836     \l_fp_input_a_decimal_int
8837     \l_fp_input_a_exponent_int
8838     \tl_set:Nx \l_fp_arg_tl
8839     {
8840       \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
8841       -
8842       \tex_else:D
8843       +
8844       \tex_fi:D
8845       \int_use:N \l_fp_input_a_integer_int
8846       .
8847       \tex_expandafter:D \use_none:n
8848       \tex_number:D \etex_numexpr:D
8849         \l_fp_input_a_decimal_int + \c_one_thousand_million
8850       e
8851       \int_use:N \l_fp_input_a_exponent_int
8852     }
8853     \tex_ifnum:D \l_fp_input_a_exponent_int < -\c_five
8854     \cs_set_protected_nopar:Npx \fp_tmp:w
8855     {
8856       \group_end:
8857       #1 \exp_not:N #2 { \l_fp_arg_tl }
8858     }
8859     \tex_else:D
8860     \etex_ifcsname:D
8861       c_fp_sin ( \l_fp_arg_tl ) _fp
8862     \tex_endcsname:D
8863     \tex_else:D

```



```

8864         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
8865         \fp_sin_aux_i:
8866     \tex_fi:D
8867     \cs_set_protected_nopar:Npx \fp_tmp:w
8868     {
8869         \group_end:
8870         #1 \exp_not:N #2
8871         { \use:c { c_fp_sin ( \l_fp_arg_tl ) _fp } }
8872     }
8873     \tex_fi:D
8874     \fp_tmp:w
8875 }

```

The internals for sine first normalise the input into an octant, then choose the correct set up for the Taylor series. The sign for the sine function is easy, so there is no worry about it. So the only thing to do is to get the output standardised.

```

8876 \cs_new_protected_nopar:Npn \fp_sin_aux_i: {
8877     \fp_trig_normalise:
8878     \fp_sin_aux_ii:
8879     \tex_ifnum:D \l_fp_output_integer_int = \c_one
8880     \l_fp_output_exponent_int \c_zero
8881     \tex_else:D
8882         \l_fp_output_integer_int \l_fp_output_decimal_int
8883         \l_fp_output_decimal_int \l_fp_output_extended_int
8884         \l_fp_output_exponent_int -\c_nine
8885     \tex_fi:D
8886     \fp_standardise:NNNN
8887     \l_fp_input_a_sign_int
8888     \l_fp_output_integer_int
8889     \l_fp_output_decimal_int
8890     \l_fp_output_exponent_int
8891     \tl_new:c { c_fp_sin ( \l_fp_arg_tl ) _fp }
8892     \tl_gset:cx { c_fp_sin ( \l_fp_arg_tl ) _fp }
8893     {
8894         \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
8895         +
8896         \tex_else:D
8897         -
8898         \tex_fi:D
8899         \int_use:N \l_fp_output_integer_int
8900         .
8901         \tex_expandafter:D \use_none:n
8902         \tex_number:D \etex_numexpr:D
8903             \l_fp_output_decimal_int + \c_one_thousand_million
8904         \scan_stop:
8905         e
8906         \int_use:N \l_fp_output_exponent_int
8907     }
8908 }

```

```

8909 \cs_new_protected_nopar:Npn \fp_sin_aux_ii: {
8910   \tex_ifcase:D \l_fp_trig_octant_int
8911   \tex_or:D
8912   \tex_expandafter:D \fp_trig_calc_sin:
8913   \tex_or:D
8914   \tex_expandafter:D \fp_trig_calc_cos:
8915   \tex_or:D
8916   \tex_expandafter:D \fp_trig_calc_cos:
8917   \tex_or:D
8918   \tex_expandafter:D \fp_trig_calc_sin:
8919   \tex_fi:D
8920 }

```

(End definition for `\fp_sin:Nn` and `\fp_sin:cn`. These functions are documented on page 155.)

```

\fp_cos:Nn Cosine is almost identical, but there is no short cut code here.
\fp_cos:cn
\fp_gcos:Nn
\fp_gcos:cn
\fp_cos_aux:NNn
\fp_cos_aux_i:
\fp_cos_aux_ii:
8921 \cs_new_protected_nopar:Npn \fp_cos:Nn {
8922   \fp_cos_aux:NNn \tl_set:Nn
8923 }
8924 \cs_new_protected_nopar:Npn \fp_gcos:Nn {
8925   \fp_cos_aux:NNn \tl_gset:Nn
8926 }
8927 \cs_generate_variant:Nn \fp_cos:Nn { c }
8928 \cs_generate_variant:Nn \fp_gcos:Nn { c }
8929 \cs_new_protected_nopar:Npn \fp_cos_aux:NNn #1#2#3 {
8930   \group_begin:
8931   \fp_split:Nn a {#3}
8932   \fp_standardise:NNNN
8933   \l_fp_input_a_sign_int
8934   \l_fp_input_a_integer_int
8935   \l_fp_input_a_decimal_int
8936   \l_fp_input_a_exponent_int
8937   \tl_set:Nx \l_fp_arg_tl
8938   {
8939     \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
8940     -
8941     \tex_else:D
8942     +
8943     \tex_fi:D
8944     \int_use:N \l_fp_input_a_integer_int
8945     .
8946     \tex_expandafter:D \use_none:n
8947     \tex_number:D \etex_numexpr:D
8948     \l_fp_input_a_decimal_int + \c_one_thousand_million
8949     e
8950     \int_use:N \l_fp_input_a_exponent_int
8951   }
8952   \etex_ifcsname:D c_fp_cos ( \l_fp_arg_tl ) _fp \tex_endcsname:D
8953   \tex_else:D

```

```

8954     \tex_expandafter:D \fp_cos_aux_i:
8955     \tex_fi:D
8956     \cs_set_protected_nopar:Npx \fp_tmp:w
8957     {
8958         \group_end:
8959         #1 \exp_not:N #2
8960         { \use:c { c_fp_cos ( \l_fp_arg_tl ) _fp } }
8961     }
8962     \fp_tmp:w
8963 }

```

Almost the same as for sine: just a bit of correction for the sign of the output.

```

8964 \cs_new_protected_nopar:Npn \fp_cos_aux_i: {
8965     \fp_trig_normalise:
8966     \fp_cos_aux_ii:
8967     \tex_ifnum:D \l_fp_output_integer_int = \c_one
8968     \l_fp_output_exponent_int \c_zero
8969     \tex_else:D
8970     \l_fp_output_integer_int \l_fp_output_decimal_int
8971     \l_fp_output_decimal_int \l_fp_output_extended_int
8972     \l_fp_output_exponent_int -\c_nine
8973     \tex_fi:D
8974     \fp_standardise:NNNN
8975     \l_fp_input_a_sign_int
8976     \l_fp_output_integer_int
8977     \l_fp_output_decimal_int
8978     \l_fp_output_exponent_int
8979     \tl_new:c { c_fp_cos ( \l_fp_arg_tl ) _fp }
8980     \tl_gset:cx { c_fp_cos ( \l_fp_arg_tl ) _fp }
8981     {
8982         \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
8983         +
8984         \tex_else:D
8985         -
8986         \tex_fi:D
8987         \int_use:N \l_fp_output_integer_int
8988         .
8989         \tex_expandafter:D \use_none:n
8990         \tex_number:D \etex_numexpr:D
8991         \l_fp_output_decimal_int + \c_one_thousand_million
8992         \scan_stop:
8993         e
8994         \int_use:N \l_fp_output_exponent_int
8995     }
8996 }
8997 \cs_new_protected_nopar:Npn \fp_cos_aux_ii: {
8998     \tex_ifcase:D \l_fp_trig_octant_int
8999     \tex_or:D
9000     \tex_expandafter:D \fp_trig_calc_cos:

```

```

9001 \tex_or:D
9002   \tex_expandafter:D \fp_trig_calc_sin:
9003 \tex_or:D
9004   \tex_expandafter:D \fp_trig_calc_sin:
9005 \tex_or:D
9006   \tex_expandafter:D \fp_trig_calc_cos:
9007 \tex_fi:D
9008 \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9009   \tex_ifnum:D \l_fp_trig_octant_int > \c_two
9010     \l_fp_input_a_sign_int \c_minus_one
9011   \tex_fi:D
9012 \tex_else:D
9013   \tex_ifnum:D \l_fp_trig_octant_int > \c_two
9014   \tex_else:D
9015     \l_fp_input_a_sign_int \c_one
9016   \tex_fi:D
9017 \tex_fi:D
9018 }

```

(End definition for `\fp_cos:Nn` and `\fp_cos:cn`. These functions are documented on page 155.)

```

\fp_trig_calc_cos: These functions actually do the calculation for sine and cosine.
\fp_trig_calc_sin:
\fp_trig_calc_Taylor:
9019 \cs_new_protected_nopar:Npn \fp_trig_calc_cos: {
9020   \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
9021     \l_fp_output_integer_int \c_one
9022     \l_fp_output_decimal_int \c_zero
9023   \tex_else:D
9024     \l_fp_trig_sign_int \c_minus_one
9025   \fp_mul:NNNNNN
9026     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9027     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9028     \l_fp_trig_decimal_int \l_fp_trig_extended_int
9029   \fp_div_integer:NNNNN
9030     \l_fp_trig_decimal_int \l_fp_trig_extended_int
9031     \c_two
9032     \l_fp_trig_decimal_int \l_fp_trig_extended_int
9033   \l_fp_count_int \c_three
9034   \tex_ifnum:D \l_fp_trig_extended_int = \c_zero
9035     \tex_ifnum:D \l_fp_trig_decimal_int = \c_zero
9036       \l_fp_output_integer_int \c_one
9037       \l_fp_output_decimal_int \c_zero
9038       \l_fp_output_extended_int \c_zero
9039     \tex_else:D
9040       \l_fp_output_integer_int \c_zero
9041       \l_fp_output_decimal_int \c_one_thousand_million
9042       \l_fp_output_extended_int \c_zero
9043     \tex_fi:D
9044   \tex_else:D
9045     \l_fp_output_integer_int \c_zero

```

```

9046     \l_fp_output_decimal_int 999999999 \scan_stop:
9047     \l_fp_output_extended_int \c_one_thousand_million
9048     \tex_fi:D
9049     \tex_advance:D \l_fp_output_extended_int -\l_fp_trig_extended_int
9050     \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
9051     \tex_expandafter:D \fp_trig_calc_Taylor:
9052     \tex_fi:D
9053 }
9054 \cs_new_protected_nopar:Npn \fp_trig_calc_sin: {
9055     \l_fp_output_integer_int \c_zero
9056     \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
9057         \l_fp_output_decimal_int \c_zero
9058     \tex_else:D
9059         \l_fp_output_decimal_int \l_fp_input_a_decimal_int
9060         \l_fp_output_extended_int \l_fp_input_a_extended_int
9061         \l_fp_trig_sign_int \c_one
9062         \l_fp_trig_decimal_int \l_fp_input_a_decimal_int
9063         \l_fp_trig_extended_int \l_fp_input_a_extended_int
9064         \l_fp_count_int \c_two
9065         \tex_expandafter:D \fp_trig_calc_Taylor:
9066     \tex_fi:D
9067 }

```

This implements a Taylor series calculation for the trigonometric functions. Lots of shuffling about as TeX is not exactly a natural choice for this sort of thing.

```

9068 \cs_new_protected_nopar:Npn \fp_trig_calc_Taylor: {
9069     \l_fp_trig_sign_int -\l_fp_trig_sign_int
9070     \fp_mul:NNNNNN
9071         \l_fp_trig_decimal_int \l_fp_trig_extended_int
9072         \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9073         \l_fp_trig_decimal_int \l_fp_trig_extended_int
9074     \fp_mul:NNNNNN
9075         \l_fp_trig_decimal_int \l_fp_trig_extended_int
9076         \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9077         \l_fp_trig_decimal_int \l_fp_trig_extended_int
9078     \fp_div_integer:NNNNN
9079         \l_fp_trig_decimal_int \l_fp_trig_extended_int
9080         \l_fp_count_int
9081         \l_fp_trig_decimal_int \l_fp_trig_extended_int
9082     \tex_advance:D \l_fp_count_int \c_one
9083     \fp_div_integer:NNNNN
9084         \l_fp_trig_decimal_int \l_fp_trig_extended_int
9085         \l_fp_count_int
9086         \l_fp_trig_decimal_int \l_fp_trig_extended_int
9087     \tex_advance:D \l_fp_count_int \c_one
9088     \tex_ifnum:D \l_fp_trig_decimal_int > \c_zero
9089         \tex_ifnum:D \l_fp_trig_sign_int > \c_zero
9090             \tex_advance:D \l_fp_output_decimal_int \l_fp_trig_decimal_int
9091             \tex_advance:D \l_fp_output_extended_int

```

```

9092         \l_fp_trig_extended_int
9093     \tex_ifnum:D \l_fp_output_extended_int < \c_one_thousand_million
9094     \tex_else:D
9095         \tex_advance:D \l_fp_output_decimal_int \c_one
9096         \tex_advance:D \l_fp_output_extended_int
9097             -\c_one_thousand_million
9098     \tex_fi:D
9099     \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
9100     \tex_else:D
9101         \tex_advance:D \l_fp_output_integer_int \c_one
9102         \tex_advance:D \l_fp_output_decimal_int
9103             -\c_one_thousand_million
9104     \tex_fi:D
9105     \tex_else:D
9106         \tex_advance:D \l_fp_output_decimal_int -\l_fp_trig_decimal_int
9107         \tex_advance:D \l_fp_output_extended_int
9108             -\l_fp_input_a_extended_int
9109     \tex_ifnum:D \l_fp_output_extended_int < \c_zero
9110         \tex_advance:D \l_fp_output_decimal_int \c_minus_one
9111         \tex_advance:D \l_fp_output_extended_int \c_one_thousand_million
9112     \tex_fi:D
9113     \tex_ifnum:D \l_fp_output_decimal_int < \c_zero
9114         \tex_advance:D \l_fp_output_integer_int \c_minus_one
9115         \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
9116     \tex_fi:D
9117     \tex_fi:D
9118     \tex_expandafter:D \fp_trig_calc_Taylor:
9119     \tex_fi:D
9120 }

```

(End definition for `\fp_trig_calc_cos:`. This function is documented on page ??.)

\fp_tan:Nn As might be expected, tangents are calculated from the sine and cosine by division. So
\fp_tan:cn there is a bit of set up, the two subsidiary pieces of work are done and then a division
\fp_gtan:Nn takes place. For small numbers, the same approach is used as for sines, with the input
\fp_gtan:cn value simply returned as is.

```

\fp_tan_aux:NNn
\fp_tan_aux_i: 9121 \cs_new_protected_nopar:Npn \fp_tan:Nn {
\fp_tan_aux_ii: 9122     \fp_tan_aux:NNn \tl_set:Nn
9123 }
\fp_tan_aux_iii: 9124 \cs_new_protected_nopar:Npn \fp_gtan:Nn {
\fp_tan_aux_iv: 9125     \fp_tan_aux:NNn \tl_gset:Nn
9126 }
9127 \cs_generate_variant:Nn \fp_tan:Nn { c }
9128 \cs_generate_variant:Nn \fp_gtan:Nn { c }
9129 \cs_new_protected_nopar:Npn \fp_tan_aux:NNn #1#2#3 {
9130     \group_begin:
9131         \fp_split:Nn a {#3}
9132         \fp_standardise:NNNN
9133         \l_fp_input_a_sign_int

```

```

9134     \l_fp_input_a_integer_int
9135     \l_fp_input_a_decimal_int
9136     \l_fp_input_a_exponent_int
9137     \tl_set:Nx \l_fp_arg_tl
9138     {
9139         \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
9140         -
9141         \tex_else:D
9142         +
9143         \tex_fi:D
9144         \int_use:N \l_fp_input_a_integer_int
9145         .
9146         \tex_expandafter:D \use_none:n
9147         \tex_number:D \etex_numexpr:D
9148             \l_fp_input_a_decimal_int + \c_one_thousand_million
9149         e
9150         \int_use:N \l_fp_input_a_exponent_int
9151     }
9152     \tex_ifnum:D \l_fp_input_a_exponent_int < -\c_five
9153     \cs_set_protected_nopar:Npx \fp_tmp:w
9154     {
9155         \group_end:
9156         #1 \exp_not:N #2 { \l_fp_arg_tl }
9157     }
9158     \tex_else:D
9159     \etex_ifcsname:D
9160         c_fp_tan ( \l_fp_arg_tl ) _fp
9161     \tex_endcsname:D
9162     \tex_else:D
9163     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9164     \fp_tan_aux_i:
9165     \tex_fi:D
9166     \cs_set_protected_nopar:Npx \fp_tmp:w
9167     {
9168         \group_end:
9169         #1 \exp_not:N #2
9170             { \use:c { c_fp_tan ( \l_fp_arg_tl ) _fp } }
9171     }
9172     \tex_fi:D
9173     \fp_tmp:w
9174 }

```

The business of the calculation does not check for stored sines or cosines as there would then be an overhead to reading them back in. There is also no need to worry about ‘small’ sine values as these will have been dealt with earlier. There is a two-step lead off so that undefined division is not even attempted.

```

9175 \cs_new_protected_nopar:Npn \fp_tan_aux_i: {
9176     \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
9177     \tex_expandafter:D \fp_tan_aux_ii:

```

```

9178 \tex_else:D
9179   \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9180   \c_zero_fp
9181   \tex_expandafter:D \fp_trig_overflow_msg:
9182 \tex_fi:D
9183 }
9184 \cs_new_protected_nopar:Npn \fp_tan_aux_ii: {
9185   \fp_trig_normalise:
9186   \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9187   \tex_ifnum:D \l_fp_trig_octant_int > \c_two
9188   \l_fp_output_sign_int \c_minus_one
9189   \tex_else:D
9190   \l_fp_output_sign_int \c_one
9191   \tex_fi:D
9192 \tex_else:D
9193   \tex_ifnum:D \l_fp_trig_octant_int > \c_two
9194   \l_fp_output_sign_int \c_one
9195   \tex_else:D
9196   \l_fp_output_sign_int \c_minus_one
9197   \tex_fi:D
9198 \tex_fi:D
9199 \fp_cos_aux_ii:
9200 \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
9201   \tex_ifnum:D \l_fp_input_a_integer_int = \c_zero
9202   \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9203   \c_undefined_fp
9204   \tex_else:D
9205   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9206   \fp_tan_aux_iii:
9207   \tex_fi:D
9208 \tex_else:D
9209   \tex_expandafter:D \fp_tan_aux_iii:
9210 \tex_fi:D
9211 }

```

The division is done here using the same code as the standard division unit, shifting the digits in the calculated sine and cosine to maintain accuracy.

```

9212 \cs_new_protected_nopar:Npn \fp_tan_aux_iii: {
9213   \l_fp_input_b_integer_int \l_fp_output_decimal_int
9214   \l_fp_input_b_decimal_int \l_fp_output_extended_int
9215   \l_fp_input_b_exponent_int -\c_nine
9216   \fp_standardise:NNNN
9217   \l_fp_input_b_sign_int
9218   \l_fp_input_b_integer_int
9219   \l_fp_input_b_decimal_int
9220   \l_fp_input_b_exponent_int
9221 \fp_sin_aux_ii:
9222 \l_fp_input_a_integer_int \l_fp_output_decimal_int
9223 \l_fp_input_a_decimal_int \l_fp_output_extended_int

```



```

9224 \l_fp_input_a_exponent_int -\c_nine
9225 \fp_standardise:NNNN
9226   \l_fp_input_a_sign_int
9227   \l_fp_input_a_integer_int
9228   \l_fp_input_a_decimal_int
9229   \l_fp_input_a_exponent_int
9230 \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
9231 \tex_ifnum:D \l_fp_input_a_integer_int = \c_zero
9232   \cs_new_eq:cN { c_fp_tan ( \l_fp_arg_tl ) _fp }
9233     \c_zero_fp
9234 \tex_else:D
9235   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9236     \fp_tan_aux_iv:
9237 \tex_fi:D
9238 \tex_else:D
9239   \tex_expandafter:D \fp_tan_aux_iv:
9240 \tex_fi:D
9241 }
9242 \cs_new_protected_nopar:Npn \fp_tan_aux_iv: {
9243   \l_fp_output_integer_int \c_zero
9244   \l_fp_output_decimal_int \c_zero
9245   \cs_set_eq:NN \fp_div_store: \fp_div_store_integer:
9246   \l_fp_div_offset_int \c_one_hundred_million
9247   \fp_div_loop:
9248   \l_fp_output_exponent_int
9249   \etex_numexpr:D
9250     \l_fp_input_a_exponent_int - \l_fp_input_b_exponent_int
9251   \scan_stop:
9252 \fp_standardise:NNNN
9253   \l_fp_output_sign_int
9254   \l_fp_output_integer_int
9255   \l_fp_output_decimal_int
9256   \l_fp_output_exponent_int
9257 \tl_new:c { c_fp_tan ( \l_fp_arg_tl ) _fp }
9258 \tl_gset:cx { c_fp_tan ( \l_fp_arg_tl ) _fp }
9259   {
9260     \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9261       +
9262     \tex_else:D
9263       -
9264     \tex_fi:D
9265     \int_use:N \l_fp_output_integer_int
9266     .
9267     \tex_expandafter:D \use_none:n
9268     \tex_number:D \etex_numexpr:D
9269       \l_fp_output_decimal_int + \c_one_thousand_million
9270     \scan_stop:
9271     e
9272     \int_use:N \l_fp_output_exponent_int
9273   }

```

9274 }

(End definition for `\fp_tan:Nn` and `\fp_tan:cn`. These functions are documented on page 155.)

118.12 Exponent and logarithm functions

Calculation of exponentials requires a number of precomputed values: first the positive integers.

```

\c_fp_exp_1_tl 9275 \tl_new:c { c_fp_exp_1_tl }
\c_fp_exp_2_tl 9276 \tl_set:cn { c_fp_exp_1_tl }
\c_fp_exp_3_tl 9277 { { 2 } { 718281828 } { 459045235 } { 0 } }
\c_fp_exp_4_tl 9278 \tl_new:c { c_fp_exp_2_tl }
\c_fp_exp_5_tl 9279 \tl_set:cn { c_fp_exp_2_tl }
\c_fp_exp_6_tl 9280 { { 7 } { 389056098 } { 930650227 } { 0 } }
\c_fp_exp_7_tl 9281 \tl_new:c { c_fp_exp_3_tl }
\c_fp_exp_8_tl 9282 \tl_set:cn { c_fp_exp_3_tl }
\c_fp_exp_9_tl 9283 { { 2 } { 008553692 } { 318766774 } { 1 } }
\c_fp_exp_10_tl 9284 \tl_new:c { c_fp_exp_4_tl }
\c_fp_exp_20_tl 9285 \tl_set:cn { c_fp_exp_4_tl }
\c_fp_exp_30_tl 9286 { { 5 } { 459815003 } { 314423908 } { 1 } }
\c_fp_exp_40_tl 9287 \tl_new:c { c_fp_exp_5_tl }
\c_fp_exp_50_tl 9288 \tl_set:cn { c_fp_exp_5_tl }
\c_fp_exp_60_tl 9289 { { 1 } { 484131591 } { 025766034 } { 2 } }
\c_fp_exp_70_tl 9290 \tl_new:c { c_fp_exp_6_tl }
\c_fp_exp_80_tl 9291 \tl_set:cn { c_fp_exp_6_tl }
\c_fp_exp_90_tl 9292 { { 4 } { 034287934 } { 927351226 } { 2 } }
\c_fp_exp_100_tl 9293 \tl_new:c { c_fp_exp_7_tl }
\c_fp_exp_200_tl 9294 \tl_set:cn { c_fp_exp_7_tl }
9295 { { 1 } { 096633158 } { 428458599 } { 3 } }
9296 \tl_new:c { c_fp_exp_8_tl }
9297 \tl_set:cn { c_fp_exp_8_tl }
9298 { { 2 } { 980957987 } { 041728275 } { 3 } }
9299 \tl_new:c { c_fp_exp_9_tl }
9300 \tl_set:cn { c_fp_exp_9_tl }
9301 { { 8 } { 103083927 } { 575384008 } { 3 } }
9302 \tl_new:c { c_fp_exp_10_tl }
9303 \tl_set:cn { c_fp_exp_10_tl }
9304 { { 2 } { 202646579 } { 480671652 } { 4 } }
9305 \tl_new:c { c_fp_exp_20_tl }
9306 \tl_set:cn { c_fp_exp_20_tl }
9307 { { 4 } { 851651954 } { 097902280 } { 8 } }
9308 \tl_new:c { c_fp_exp_30_tl }
9309 \tl_set:cn { c_fp_exp_30_tl }
9310 { { 1 } { 068647458 } { 152446215 } { 13 } }
9311 \tl_new:c { c_fp_exp_40_tl }
9312 \tl_set:cn { c_fp_exp_40_tl }
9313 { { 2 } { 353852668 } { 370199854 } { 17 } }
9314 \tl_new:c { c_fp_exp_50_tl }

```

```

9315 \tl_set:cn { c_fp_exp_50_tl }
9316   { { 5 } { 184705528 } { 587072464 } { 21 } }
9317 \tl_new:c { c_fp_exp_60_tl }
9318 \tl_set:cn { c_fp_exp_60_tl }
9319   { { 1 } { 142007389 } { 815684284 } { 26 } }
9320 \tl_new:c { c_fp_exp_70_tl }
9321 \tl_set:cn { c_fp_exp_70_tl }
9322   { { 2 } { 515438670 } { 919167006 } { 30 } }
9323 \tl_new:c { c_fp_exp_80_tl }
9324 \tl_set:cn { c_fp_exp_80_tl }
9325   { { 5 } { 540622384 } { 393510053 } { 34 } }
9326 \tl_new:c { c_fp_exp_90_tl }
9327 \tl_set:cn { c_fp_exp_90_tl }
9328   { { 1 } { 220403294 } { 317840802 } { 39 } }
9329 \tl_new:c { c_fp_exp_100_tl }
9330 \tl_set:cn { c_fp_exp_100_tl }
9331   { { 2 } { 688117141 } { 816135448 } { 43 } }
9332 \tl_new:c { c_fp_exp_200_tl }
9333 \tl_set:cn { c_fp_exp_200_tl }
9334   { { 7 } { 225973768 } { 125749258 } { 86 } }

```

(End definition for `\c_fp_exp_1_tl`.)

\c_fp_exp_1_tl Now the negative integers.

```

\c_fp_exp_2_tl
\c_fp_exp_3_tl
\c_fp_exp_4_tl
\c_fp_exp_5_tl
\c_fp_exp_6_tl
\c_fp_exp_7_tl
\c_fp_exp_8_tl
\c_fp_exp_9_tl
\c_fp_exp_10_tl
\c_fp_exp_20_tl
\c_fp_exp_30_tl
\c_fp_exp_40_tl
\c_fp_exp_50_tl
\c_fp_exp_60_tl
\c_fp_exp_70_tl
\c_fp_exp_80_tl
\c_fp_exp_90_tl
\c_fp_exp_100_tl
\c_fp_exp_200_tl

```

```

9335 \tl_new:c { c_fp_exp_-1_tl }
9336 \tl_set:cn { c_fp_exp_-1_tl }
9337   { { 3 } { 678794411 } { 71442322 } { -1 } }
9338 \tl_new:c { c_fp_exp_-2_tl }
9339 \tl_set:cn { c_fp_exp_-2_tl }
9340   { { 1 } { 353352832 } { 366132692 } { -1 } }
9341 \tl_new:c { c_fp_exp_-3_tl }
9342 \tl_set:cn { c_fp_exp_-3_tl }
9343   { { 4 } { 978706836 } { 786394298 } { -2 } }
9344 \tl_new:c { c_fp_exp_-4_tl }
9345 \tl_set:cn { c_fp_exp_-4_tl }
9346   { { 1 } { 831563888 } { 873418029 } { -2 } }
9347 \tl_new:c { c_fp_exp_-5_tl }
9348 \tl_set:cn { c_fp_exp_-5_tl }
9349   { { 6 } { 737946999 } { 085467097 } { -3 } }
9350 \tl_new:c { c_fp_exp_-6_tl }
9351 \tl_set:cn { c_fp_exp_-6_tl }
9352   { { 2 } { 478752176 } { 666358423 } { -3 } }
9353 \tl_new:c { c_fp_exp_-7_tl }
9354 \tl_set:cn { c_fp_exp_-7_tl }
9355   { { 9 } { 118819655 } { 545162080 } { -4 } }
9356 \tl_new:c { c_fp_exp_-8_tl }
9357 \tl_set:cn { c_fp_exp_-8_tl }
9358   { { 3 } { 354626279 } { 025118388 } { -4 } }
9359 \tl_new:c { c_fp_exp_-9_tl }

```

```

9360 \tl_set:cn { c_fp_exp_-9_tl }
9361 { { 1 } { 234098040 } { 866795495 } { -4 } }
9362 \tl_new:c { c_fp_exp_-10_tl }
9363 \tl_set:cn { c_fp_exp_-10_tl }
9364 { { 4 } { 539992976 } { 248451536 } { -5 } }
9365 \tl_new:c { c_fp_exp_-20_tl }
9366 \tl_set:cn { c_fp_exp_-20_tl }
9367 { { 2 } { 061153622 } { 438557828 } { -9 } }
9368 \tl_new:c { c_fp_exp_-30_tl }
9369 \tl_set:cn { c_fp_exp_-30_tl }
9370 { { 9 } { 357622968 } { 840174605 } { -14 } }
9371 \tl_new:c { c_fp_exp_-40_tl }
9372 \tl_set:cn { c_fp_exp_-40_tl }
9373 { { 4 } { 248354255 } { 291588995 } { -18 } }
9374 \tl_new:c { c_fp_exp_-50_tl }
9375 \tl_set:cn { c_fp_exp_-50_tl }
9376 { { 1 } { 928749847 } { 963917783 } { -22 } }
9377 \tl_new:c { c_fp_exp_-60_tl }
9378 \tl_set:cn { c_fp_exp_-60_tl }
9379 { { 8 } { 756510762 } { 696520338 } { -27 } }
9380 \tl_new:c { c_fp_exp_-70_tl }
9381 \tl_set:cn { c_fp_exp_-70_tl }
9382 { { 3 } { 975449735 } { 908646808 } { -31 } }
9383 \tl_new:c { c_fp_exp_-80_tl }
9384 \tl_set:cn { c_fp_exp_-80_tl }
9385 { { 1 } { 804851387 } { 845415172 } { -35 } }
9386 \tl_new:c { c_fp_exp_-90_tl }
9387 \tl_set:cn { c_fp_exp_-90_tl }
9388 { { 8 } { 194012623 } { 990515430 } { -40 } }
9389 \tl_new:c { c_fp_exp_-100_tl }
9390 \tl_set:cn { c_fp_exp_-100_tl }
9391 { { 3 } { 720075976 } { 020835963 } { -44 } }
9392 \tl_new:c { c_fp_exp_-200_tl }
9393 \tl_set:cn { c_fp_exp_-200_tl }
9394 { { 1 } { 383896526 } { 736737530 } { -87 } }

```

(End definition for `\c_fp_exp_-1_tl`.)

`\fp_exp:Nn` The calculation of an exponent starts off starts in much the same way as the trigonometric
`\fp_exp:cn` functions: normalise the input, look for a pre-defined value and if one is not found hand
`\fp_gexp:Nn` off to the real workhorse function. The test for a definition of the result is used so that
`\fp_gexp:cn` overflows do not result in any outcome being defined.

```

\fp_exp_aux:NNn
\fp_exp_internal:
  \fp_exp_aux:
    \fp_exp_integer:
      \fp_exp_integer_tens:
        \fp_exp_integer_units:
          \fp_exp_integer_const:n
\fp_exp_integer_const:nnnn
  \fp_exp_decimal:
  \fp_exp_Taylor:
  \fp_exp_const:Nx
  \fp_exp_const:cx

```

```

9402 \cs_generate_variant:Nn \fp_gexp:Nn { c }
9403 \cs_new_protected_nopar:Npn \fp_exp_aux:NNn #1#2#3 {
9404   \group_begin:
9405     \fp_split:Nn a {#3}
9406     \fp_standardise:NNNN
9407     \l_fp_input_a_sign_int
9408     \l_fp_input_a_integer_int
9409     \l_fp_input_a_decimal_int
9410     \l_fp_input_a_exponent_int
9411     \l_fp_input_a_extended_int \c_zero
9412     \tl_set:Nx \l_fp_arg_tl
9413     {
9414       \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
9415       -
9416       \tex_else:D
9417       +
9418       \tex_fi:D
9419       \int_use:N \l_fp_input_a_integer_int
9420       .
9421       \tex_expandafter:D \use_none:n
9422       \tex_number:D \etex_numexpr:D
9423         \l_fp_input_a_decimal_int + \c_one_thousand_million
9424       e
9425       \int_use:N \l_fp_input_a_exponent_int
9426     }
9427     \etex_ifcsname:D c_fp_exp ( \l_fp_arg_tl ) _fp \tex_endcsname:D
9428     \tex_else:D
9429     \tex_expandafter:D \fp_exp_internal:
9430     \tex_fi:D
9431     \cs_set_protected_nopar:Npx \fp_tmp:w
9432     {
9433       \group_end:
9434       #1 \exp_not:N #2
9435       {
9436         \etex_ifcsname:D c_fp_exp ( \l_fp_arg_tl ) _fp
9437         \tex_endcsname:D
9438         \use:c { c_fp_exp ( \l_fp_arg_tl ) _fp }
9439         \tex_else:D
9440         \c_zero_fp
9441         \tex_fi:D
9442       }
9443     }
9444     \fp_tmp:w
9445 }

```

The first real step is to convert the input into a fixed-point representation for further calculation: anything which is dropped here as too small would not influence the output in any case. There are a couple of overflow tests: the maximum

```

9446 \cs_new_protected_nopar:Npn \fp_exp_internal: {

```

```

9447 \tex_ifnum:D \l_fp_input_a_exponent_int < \c_three
9448 \fp_extended_normalise:
9449 \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9450 \tex_ifnum:D \l_fp_input_a_integer_int < 230 \scan_stop:
9451 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9452 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9453 \tex_expandafter:D \fp_exp_aux:
9454 \tex_else:D
9455 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9456 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9457 \tex_expandafter:D \fp_exp_overflow_msg:
9458 \tex_fi:D
9459 \tex_else:D
9460 \tex_ifnum:D \l_fp_input_a_integer_int < 230 \scan_stop:
9461 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9462 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9463 \tex_expandafter:D \fp_exp_aux:
9464 \tex_else:D
9465 \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
9466 { \c_zero_fp }
9467 \tex_fi:D
9468 \tex_fi:D
9469 \tex_else:D
9470 \tex_expandafter:D \fp_exp_overflow_msg:
9471 \tex_fi:D
9472 }

```

The main algorithm makes use of the fact that

$$e^{nmp.q} = e^n e^m e^p e^{0.q}$$

and that there is a Taylor series that can be used to calculate $e^{0.q}$. Thus the approach needed is in three parts. First, the exponent of the integer part of the input is found using the pre-calculated constants. Second, the Taylor series is used to find the exponent for the decimal part of the input. Finally, the two parts are multiplied together to give the result. As the normalisation code will already have dealt with any overflowing values, there are no further checks needed.

```

9473 \cs_new_protected_nopar:Npn \fp_exp_aux: {
9474 \tex_ifnum:D \l_fp_input_a_integer_int > \c_zero
9475 \tex_expandafter:D \fp_exp_integer:
9476 \tex_else:D
9477 \l_fp_output_integer_int \c_one
9478 \l_fp_output_decimal_int \c_zero
9479 \l_fp_output_extended_int \c_zero
9480 \l_fp_output_exponent_int \c_zero
9481 \tex_expandafter:D \fp_exp_decimal:
9482 \tex_fi:D
9483 }

```

The integer part calculation starts with the hundreds. This is set up such that very large negative numbers can short-cut the entire procedure and simply return zero. In other cases, the code either recovers the exponent of the hundreds value or sets the appropriate storage to one (so that multiplication works correctly).

```

9484 \cs_new_protected_nopar:Npn \fp_exp_integer: {
9485   \tex_ifnum:D \l_fp_input_a_integer_int < \c_one_hundred
9486     \l_fp_exp_integer_int \c_one
9487     \l_fp_exp_decimal_int \c_zero
9488     \l_fp_exp_extended_int \c_zero
9489     \l_fp_exp_exponent_int \c_zero
9490   \tex_expandafter:D \fp_exp_integer_tens:
9491 \tex_else:D
9492   \tl_set:Nx \l_fp_tmp_tl
9493     {
9494       \tex_expandafter:D \use_i:nnn
9495       \int_use:N \l_fp_input_a_integer_int
9496     }
9497   \l_fp_input_a_integer_int
9498   \etex_numexpr:D
9499     \l_fp_input_a_integer_int - \l_fp_tmp_tl 00
9500   \scan_stop:
9501 \tex_ifnum:D \l_fp_input_a_sign_int < \c_zero
9502 \tex_ifnum:D \l_fp_output_integer_int > 200 \scan_stop:
9503   \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
9504   { \c_zero_fp }
9505 \tex_else:D
9506   \fp_exp_integer_const:n { - \l_fp_tmp_tl 00 }
9507   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9508   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9509   \tex_expandafter:D \fp_exp_integer_tens:
9510   \tex_fi:D
9511 \tex_else:D
9512   \fp_exp_integer_const:n { \l_fp_tmp_tl 00 }
9513   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9514   \tex_expandafter:D \fp_exp_integer_tens:
9515   \tex_fi:D
9516 \tex_fi:D
9517 }

```

The tens and units parts are handled in a similar way, with a multiplication step to build up the final value. That also includes a correction step to avoid an overflow of the integer part.

```

9518 \cs_new_protected_nopar:Npn \fp_exp_integer_tens: {
9519   \l_fp_output_integer_int \l_fp_exp_integer_int
9520   \l_fp_output_decimal_int \l_fp_exp_decimal_int
9521   \l_fp_output_extended_int \l_fp_exp_extended_int
9522   \l_fp_output_exponent_int \l_fp_exp_exponent_int
9523   \tex_ifnum:D \l_fp_input_a_integer_int > \c_nine

```

```

9524 \tl_set:Nx \l_fp_tmp_tl
9525 {
9526   \tex_expandafter:D \use_i:nn
9527   \int_use:N \l_fp_input_a_integer_int
9528 }
9529 \l_fp_input_a_integer_int
9530 \etex_numexpr:D
9531   \l_fp_input_a_integer_int - \l_fp_tmp_tl 0
9532 \scan_stop:
9533 \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9534   \fp_exp_integer_const:n { \l_fp_tmp_tl 0 }
9535 \tex_else:D
9536   \fp_exp_integer_const:n { - \l_fp_tmp_tl 0 }
9537 \tex_fi:D
9538 \fp_mul:NNNNNNNN
9539   \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9540   \l_fp_output_integer_int \l_fp_output_decimal_int
9541   \l_fp_output_extended_int
9542   \l_fp_output_integer_int \l_fp_output_decimal_int
9543   \l_fp_output_extended_int
9544   \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
9545   \fp_extended_normalise_output:
9546 \tex_fi:D
9547 \fp_exp_integer_units:
9548 }
9549 \cs_new_protected_nopar:Npn \fp_exp_integer_units: {
9550   \tex_ifnum:D \l_fp_input_a_integer_int > \c_zero
9551     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9552       \fp_exp_integer_const:n { \int_use:N \l_fp_input_a_integer_int }
9553   \tex_else:D
9554     \fp_exp_integer_const:n
9555       { - \int_use:N \l_fp_input_a_integer_int }
9556   \tex_fi:D
9557   \fp_mul:NNNNNNNN
9558     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9559     \l_fp_output_integer_int \l_fp_output_decimal_int
9560     \l_fp_output_extended_int
9561     \l_fp_output_integer_int \l_fp_output_decimal_int
9562     \l_fp_output_extended_int
9563     \tex_advance:D \l_fp_output_exponent_int \l_fp_exp_exponent_int
9564     \fp_extended_normalise_output:
9565   \tex_fi:D
9566   \fp_exp_decimal:
9567 }

```

Recovery of the stored constant values into the separate registers is done with a simple expansion then assignment.

```

9568 \cs_new_protected_nopar:Npn \fp_exp_integer_const:n #1 {
9569   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D

```



```

9570 \fp_exp_integer_const:nnnn
9571 \tex_csname:D c_fp_exp_ #1 _tl \tex_endcsname:D
9572 }
9573 \cs_new_protected_nopar:Npn \fp_exp_integer_const:nnnn #1#2#3#4 {
9574 \l_fp_exp_integer_int #1 \scan_stop:
9575 \l_fp_exp_decimal_int #2 \scan_stop:
9576 \l_fp_exp_extended_int #3 \scan_stop:
9577 \l_fp_exp_exponent_int #4 \scan_stop:
9578 }

```

Finding the exponential for the decimal part of the number requires a Taylor series calculation. The set up is done here with the loop itself a separate function. Once the decimal part is available this is multiplied by the integer part already worked out to give the final result.

```

9579 \cs_new_protected_nopar:Npn \fp_exp_decimal: {
9580 \tex_ifnum:D \l_fp_input_a_decimal_int > \c_zero
9581 \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9582 \l_fp_exp_integer_int \c_one
9583 \l_fp_exp_decimal_int \l_fp_input_a_decimal_int
9584 \l_fp_exp_extended_int \l_fp_input_a_extended_int
9585 \tex_else:D
9586 \l_fp_exp_integer_int \c_zero
9587 \tex_ifnum:D \l_fp_exp_extended_int = \c_zero
9588 \l_fp_exp_decimal_int
9589 \etex_numexpr:D
9590 \c_one_thousand_million - \l_fp_input_a_decimal_int
9591 \scan_stop:
9592 \l_fp_exp_extended_int \c_zero
9593 \tex_else:D
9594 \l_fp_exp_decimal_int
9595 \etex_numexpr:D
9596 999999999 - \l_fp_input_a_decimal_int
9597 \scan_stop:
9598 \l_fp_exp_extended_int
9599 \etex_numexpr:D
9600 \c_one_thousand_million - \l_fp_input_a_extended_int
9601 \scan_stop:
9602 \tex_fi:D
9603 \tex_fi:D
9604 \l_fp_input_b_sign_int \l_fp_input_a_sign_int
9605 \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
9606 \l_fp_input_b_extended_int \l_fp_input_a_extended_int
9607 \l_fp_count_int \c_one
9608 \fp_exp_Taylor:
9609 \fp_mul:NNNNNNNNN
9610 \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9611 \l_fp_output_integer_int \l_fp_output_decimal_int
9612 \l_fp_output_extended_int
9613 \l_fp_output_integer_int \l_fp_output_decimal_int

```

```

9614         \l_fp_output_extended_int
9615 \tex_fi:D
9616 \tex_ifnum:D \l_fp_output_extended_int < \c_five_hundred_million
9617 \tex_else:D
9618   \tex_advance:D \l_fp_output_decimal_int \c_one
9619   \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
9620   \tex_else:D
9621     \l_fp_output_decimal_int \c_zero
9622     \tex_advance:D \l_fp_output_integer_int \c_one
9623   \tex_fi:D
9624 \tex_fi:D
9625 \fp_standardise:NNNN
9626   \l_fp_output_sign_int
9627   \l_fp_output_integer_int
9628   \l_fp_output_decimal_int
9629   \l_fp_output_exponent_int
9630 \fp_exp_const:cx { c_fp_exp ( \l_fp_arg_tl ) _fp }
9631 {
9632   +
9633   \int_use:N \l_fp_output_integer_int
9634   .
9635   \tex_expandafter:D \use_none:n
9636   \tex_number:D \etex_numexpr:D
9637     \l_fp_output_decimal_int + \c_one_thousand_million
9638   \scan_stop:
9639   e
9640   \int_use:N \l_fp_output_exponent_int
9641 }
9642 }

```

The Taylor series for $\exp(x)$ is

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

which converges for $-1 < x < 1$. The code above sets up the x part, leaving the loop to multiply the running value by x/n and add it onto the sum. The way that this is done is that the running total is stored in the `exp` set of registers, while the current item is stored as `input_b`.

```

9643 \cs_new_protected_nopar:Npn \fp_exp_Taylor: {
9644   \tex_advance:D \l_fp_count_int \c_one
9645   \tex_multiply:D \l_fp_input_b_sign_int \l_fp_input_a_sign_int
9646   \fp_mul:NNNNNN
9647     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9648     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9649     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9650 \fp_div_integer:NNNNN
9651   \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9652   \l_fp_count_int

```

```

9653 \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9654 \tex_ifnum:D
9655 \etex_numexpr:D
9656 \l_fp_input_b_decimal_int + \l_fp_input_b_extended_int
9657 > \c_zero
9658 \tex_ifnum:D \l_fp_input_b_sign_int > \c_zero
9659 \tex_advance:D \l_fp_exp_decimal_int \l_fp_input_b_decimal_int
9660 \tex_advance:D \l_fp_exp_extended_int
9661 \l_fp_input_b_extended_int
9662 \tex_ifnum:D \l_fp_exp_extended_int < \c_one_thousand_million
9663 \tex_else:D
9664 \tex_advance:D \l_fp_exp_decimal_int \c_one
9665 \tex_advance:D \l_fp_exp_extended_int
9666 -\c_one_thousand_million
9667 \tex_fi:D
9668 \tex_ifnum:D \l_fp_exp_decimal_int < \c_one_thousand_million
9669 \tex_else:D
9670 \tex_advance:D \l_fp_exp_integer_int \c_one
9671 \tex_advance:D \l_fp_exp_decimal_int
9672 -\c_one_thousand_million
9673 \tex_fi:D
9674 \tex_else:D
9675 \tex_advance:D \l_fp_exp_decimal_int -\l_fp_input_b_decimal_int
9676 \tex_advance:D \l_fp_exp_extended_int
9677 -\l_fp_input_a_extended_int
9678 \tex_ifnum:D \l_fp_exp_extended_int < \c_zero
9679 \tex_advance:D \l_fp_exp_decimal_int \c_minus_one
9680 \tex_advance:D \l_fp_exp_extended_int \c_one_thousand_million
9681 \tex_fi:D
9682 \tex_ifnum:D \l_fp_exp_decimal_int < \c_zero
9683 \tex_advance:D \l_fp_exp_integer_int \c_minus_one
9684 \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
9685 \tex_fi:D
9686 \tex_fi:D
9687 \tex_expandafter:D \fp_exp_Taylor:
9688 \tex_fi:D
9689 }

```

This is set up as a function so that the power code can redirect the effect.

```

9690 \cs_new_protected_nopar:Npn \fp_exp_const:Nx #1#2 {
9691 \tl_new:N #1
9692 \tl_gset:Nx #1 {#2}
9693 }
9694 \cs_generate_variant:Nn \fp_exp_const:Nx { c }

```

(End definition for `\fp_exp:Nn` and `\fp_exp:cn`. These functions are documented on page 154.)

```

\c_fp_ln_10_1_tl
\c_fp_ln_10_2_tl
\c_fp_ln_10_3_tl
\c_fp_ln_10_4_tl
\c_fp_ln_10_5_tl
\c_fp_ln_10_6_tl
\c_fp_ln_10_7_tl
\c_fp_ln_10_8_tl
\c_fp_ln_10_9_tl

```

Constants for working out logarithms: first those for the powers of ten.

```

9695 \tl_new:c { c_fp_ln_10_1_tl }
9696 \tl_set:cn { c_fp_ln_10_1_tl }
9697   { { 2 } { 302585092 } { 994045684 } { 0 } }
9698 \tl_new:c { c_fp_ln_10_2_tl }
9699 \tl_set:cn { c_fp_ln_10_2_tl }
9700   { { 4 } { 605170185 } { 988091368 } { 0 } }
9701 \tl_new:c { c_fp_ln_10_3_tl }
9702 \tl_set:cn { c_fp_ln_10_3_tl }
9703   { { 6 } { 907755278 } { 982137052 } { 0 } }
9704 \tl_new:c { c_fp_ln_10_4_tl }
9705 \tl_set:cn { c_fp_ln_10_4_tl }
9706   { { 9 } { 210340371 } { 976182736 } { 0 } }
9707 \tl_new:c { c_fp_ln_10_5_tl }
9708 \tl_set:cn { c_fp_ln_10_5_tl }
9709   { { 1 } { 151292546 } { 497022842 } { 1 } }
9710 \tl_new:c { c_fp_ln_10_6_tl }
9711 \tl_set:cn { c_fp_ln_10_6_tl }
9712   { { 1 } { 381551055 } { 796427410 } { 1 } }
9713 \tl_new:c { c_fp_ln_10_7_tl }
9714 \tl_set:cn { c_fp_ln_10_7_tl }
9715   { { 1 } { 611809565 } { 095831979 } { 1 } }
9716 \tl_new:c { c_fp_ln_10_8_tl }
9717 \tl_set:cn { c_fp_ln_10_8_tl }
9718   { { 1 } { 842068074 } { 395226547 } { 1 } }
9719 \tl_new:c { c_fp_ln_10_9_tl }
9720 \tl_set:cn { c_fp_ln_10_9_tl }
9721   { { 2 } { 072326583 } { 694641116 } { 1 } }

```

(End definition for `\c_fp_ln_10_1_tl`.)

`\c_fp_ln_2_1_tl` The smaller set for powers of two.

```

\c_fp_ln_2_2_tl
\c_fp_ln_2_3_tl
9722 \tl_new:c { c_fp_ln_2_1_tl }
9723 \tl_set:cn { c_fp_ln_2_1_tl }
9724   { { 0 } { 693147180 } { 559945309 } { 0 } }
9725 \tl_new:c { c_fp_ln_2_2_tl }
9726 \tl_set:cn { c_fp_ln_2_2_tl }
9727   { { 1 } { 386294361 } { 119890618 } { 0 } }
9728 \tl_new:c { c_fp_ln_2_3_tl }
9729 \tl_set:cn { c_fp_ln_2_3_tl }
9730   { { 2 } { 079441541 } { 679835928 } { 0 } }

```

(End definition for `\c_fp_ln_2_1_tl`.)

\fp_ln:Nn The approach for logarithms is again based on a mix of tables and Taylor series. Here,
\fp_ln:cn the initial validation is a bit easier and so it is set up earlier, meaning less need to escape
\fp_gln:Nn later on.
\fp_gln:cn

```

\fp_ln_aux:NNn 9731 \cs_new_protected_nopar:Npn \fp_ln:Nn {
\fp_ln_aux: 9732 \fp_ln_aux:NNn \tl_set:Nn

```

`\fp_ln_exponent:`

`\fp_ln_internal:`

477

`\fp_ln_exponent_units:`

`\fp_ln_normalise:`

`\fp_ln_normalise_aux:NNNNNNNN`

`\fp_ln_mantissa:`

`\fp_ln_mantissa_aux:`

`\fp_ln_mantissa_divide_two:`

`\fp_ln_integer_const:nn`

```

9733 }
9734 \cs_new_protected_nopar:Npn \fp_gln:Nn {
9735   \fp_ln_aux:NNn \tl_gset:Nn
9736 }
9737 \cs_generate_variant:Nn \fp_ln:Nn { c }
9738 \cs_generate_variant:Nn \fp_gln:Nn { c }
9739 \cs_new_protected_nopar:Npn \fp_ln_aux:NNn #1#2#3 {
9740   \group_begin:
9741     \fp_split:Nn a {#3}
9742     \fp_standardise:NNNN
9743     \l_fp_input_a_sign_int
9744     \l_fp_input_a_integer_int
9745     \l_fp_input_a_decimal_int
9746     \l_fp_input_a_exponent_int
9747     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
9748     \tex_ifnum:D
9749       \etex_numexpr:D
9750         \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
9751         > \c_zero
9752       \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9753         \fp_ln_aux:
9754     \tex_else:D
9755       \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
9756       {
9757         \group_end:
9758         ##1 \exp_not:N ##2 { \c_zero_fp }
9759       }
9760       \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9761         \fp_ln_error_msg:
9762       \tex_fi:D
9763     \tex_else:D
9764       \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
9765       {
9766         \group_end:
9767         ##1 \exp_not:N ##2 { \c_zero_fp }
9768       }
9769       \tex_expandafter:D \fp_ln_error_msg:
9770       \tex_fi:D
9771     \fp_tmp:w #1 #2
9772 }

```

As the input at this stage meets the validity criteria above, the argument can now be saved for further processing. There is no need to look at the sign of the input as it must be positive. The function here simply sets up to either do the full calculation or recover the stored value, as appropriate.

```

9773 \cs_new_protected_nopar:Npn \fp_ln_aux: {
9774   \tl_set:Nx \l_fp_arg_tl
9775   {
9776     +

```

```

9777     \int_use:N \l_fp_input_a_integer_int
9778     .
9779     \tex_expandafter:D \use_none:n
9780     \tex_number:D \etex_numexpr:D
9781     \l_fp_input_a_decimal_int + \c_one_thousand_million
9782     e
9783     \int_use:N \l_fp_input_a_exponent_int
9784   }
9785   \etex_ifcsname:D c_fp_ln ( \l_fp_arg_tl ) _fp \tex_endcsname:D
9786   \tex_else:D
9787     \tex_expandafter:D \fp_ln_exponent:
9788   \tex_fi:D
9789   \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
9790   {
9791     \group_end:
9792     ##1 \exp_not:N ##2
9793     { \use:c { c_fp_ln ( \l_fp_arg_tl ) _fp } }
9794   }
9795 }

```

The main algorithm here uses the fact the logarithm can be divided up, first taking out the powers of ten, then powers of two and finally using a Taylor series for the remainder.

$$\ln(10^n \times 2^m \times x) = \ln(10^n) \times \ln(2^m) \times \ln(x)$$

The second point to remember is that

$$\ln(x^{-1}) = -\ln(x)$$

which means that for the powers of 10 and 2 constants are only needed for positive powers.

The first step is to set up the sign for the output functions and work out the powers of ten in the exponent. First the larger powers are sorted out. The values for the constants are the same as those for the smaller ones, just with a shift in the exponent.

```

9796 \cs_new_protected_nopar:Npn \fp_ln_exponent: {
9797   \fp_ln_internal:
9798   \tex_ifnum:D \l_fp_output_extended_int < \c_five_hundred_million
9799   \tex_else:D
9800     \tex_advance:D \l_fp_output_decimal_int \c_one
9801     \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
9802     \tex_else:D
9803       \l_fp_output_decimal_int \c_zero
9804     \tex_advance:D \l_fp_output_integer_int \c_one
9805   \tex_fi:D
9806 \tex_fi:D
9807 \fp_standardise:NNNN
9808   \l_fp_output_sign_int
9809   \l_fp_output_integer_int

```

```

9810 \l_fp_output_decimal_int
9811 \l_fp_output_exponent_int
9812 \tl_new:c { c_fp_ln ( \l_fp_arg_tl ) _fp }
9813 \tl_gset:cx { c_fp_ln ( \l_fp_arg_tl ) _fp }
9814 {
9815   \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9816   +
9817   \tex_else:D
9818   -
9819   \tex_fi:D
9820   \int_use:N \l_fp_output_integer_int
9821   .
9822   \tex_expandafter:D \use_none:n
9823   \tex_number:D \etex_numexpr:D
9824     \l_fp_output_decimal_int + \c_one_thousand_million
9825   \scan_stop:
9826   e
9827   \int_use:N \l_fp_output_exponent_int
9828 }
9829 }
9830 \cs_new_protected_nopar:Npn \fp_ln_internal: {
9831   \tex_ifnum:D \l_fp_input_a_exponent_int < \c_zero
9832     \l_fp_input_a_exponent_int - \l_fp_input_a_exponent_int
9833     \l_fp_output_sign_int \c_minus_one
9834   \tex_else:D
9835     \l_fp_output_sign_int \c_one
9836   \tex_fi:D
9837   \tex_ifnum:D \l_fp_input_a_exponent_int > \c_nine
9838     \tl_set:Nx \l_fp_tmp_tl
9839     {
9840       \tex_expandafter:D \use_i:nn
9841       \int_use:N \l_fp_input_a_exponent_int
9842     }
9843     \l_fp_input_a_exponent_int
9844     \etex_numexpr:D
9845       \l_fp_input_a_exponent_int - \l_fp_tmp_tl 0
9846     \scan_stop:
9847     \fp_ln_const:nn { 10 } { \l_fp_tmp_tl }
9848     \tex_advance:D \l_fp_exp_exponent_int \c_one
9849     \l_fp_output_integer_int \l_fp_exp_integer_int
9850     \l_fp_output_decimal_int \l_fp_exp_decimal_int
9851     \l_fp_output_extended_int \l_fp_exp_extended_int
9852     \l_fp_output_exponent_int \l_fp_exp_exponent_int
9853   \tex_else:D
9854     \l_fp_output_integer_int \c_zero
9855     \l_fp_output_decimal_int \c_zero
9856     \l_fp_output_extended_int \c_zero
9857     \l_fp_output_exponent_int \c_zero
9858   \tex_fi:D
9859   \fp_ln_exponent_units:

```

```
9860 }
```

Next the smaller powers of ten, which will need to be combined with the above: always an additive process.

```
9861 \cs_new_protected_nopar:Npn \fp_ln_exponent_units: {
9862   \tex_ifnum:D \l_fp_input_a_exponent_int > \c_zero
9863     \fp_ln_const:nn { 10 } { \int_use:N \l_fp_input_a_exponent_int }
9864   \fp_ln_normalise:
9865   \fp_add:NNNNNNNNN
9866     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9867     \l_fp_output_integer_int \l_fp_output_decimal_int
9868     \l_fp_output_extended_int
9869     \l_fp_output_integer_int \l_fp_output_decimal_int
9870     \l_fp_output_extended_int
9871   \tex_fi:D
9872   \fp_ln_mantissa:
9873 }
```

The smaller table-based parts may need to be exponent shifted so that they stay in line with the larger parts. This is similar to the approach in other places, but here there is a need to watch the extended part of the number.

```
9874 \cs_new_protected_nopar:Npn \fp_ln_normalise: {
9875   \tex_ifnum:D \l_fp_exp_exponent_int < \l_fp_output_exponent_int
9876     \tex_advance:D \l_fp_exp_decimal_int \c_one_thousand_million
9877     \tex_expandafter:D \use_i:nn \tex_expandafter:D
9878       \fp_ln_normalise_aux:NNNNNNNNN
9879       \int_use:N \l_fp_exp_decimal_int
9880     \tex_expandafter:D \fp_ln_normalise:
9881   \tex_fi:D
9882 }
9883 \cs_new_protected_nopar:Npn
9884   \fp_ln_normalise_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
9885   \tex_ifnum:D \l_fp_exp_integer_int = \c_zero
9886     \l_fp_exp_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
9887   \tex_else:D
9888     \tl_set:Nx \l_fp_tmp_tl
9889     {
9890       \int_use:N \l_fp_exp_integer_int
9891       #1#2#3#4#5#6#7#8
9892     }
9893     \l_fp_exp_integer_int \c_zero
9894     \l_fp_exp_decimal_int \l_fp_tmp_tl \scan_stop:
9895   \tex_fi:D
9896   \tex_divide:D \l_fp_exp_extended_int \c_ten
9897   \tl_set:Nx \l_fp_tmp_tl
9898   {
9899     #9
9900     \int_use:N \l_fp_exp_extended_int
```



```

9901     }
9902     \l_fp_exp_extended_int \l_fp_tmp_tl \scan_stop:
9903     \tex_advance:D \l_fp_exp_exponent_int \c_one
9904 }

```

The next phase is to decompose the mantissa by division by two to leave a value which is in the range $1 \leq x < 2$. The sum of the two powers needs to take account of the sign of the output: if it is negative then the result gets *smaller* as the mantissa gets *bigger*.

```

9905 \cs_new_protected_nopar:Npn \fp_ln_mantissa: {
9906   \l_fp_count_int \c_zero
9907   \l_fp_input_a_extended_int \c_zero
9908   \fp_ln_mantissa_aux:
9909   \tex_ifnum:D \l_fp_count_int > \c_zero
9910     \fp_ln_const:nn { 2 } { \int_use:N \l_fp_count_int }
9911     \fp_ln_normalise:
9912     \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9913       \tex_expandafter:D \fp_add:NNNNNNNNN
9914     \tex_else:D
9915       \tex_expandafter:D \fp_sub:NNNNNNNNN
9916     \tex_fi:D
9917     \l_fp_output_integer_int \l_fp_output_decimal_int
9918     \l_fp_output_extended_int
9919     \l_fp_exp_integer_int \l_fp_exp_decimal_int \l_fp_exp_extended_int
9920     \l_fp_output_integer_int \l_fp_output_decimal_int
9921     \l_fp_output_extended_int
9922   \tex_fi:D
9923   \tex_ifnum:D
9924     \etex_numexpr:D
9925       \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int > \c_one
9926     \scan_stop:
9927     \tex_expandafter:D \fp_ln_Taylor:
9928   \tex_fi:D
9929 }
9930 \cs_new_protected_nopar:Npn \fp_ln_mantissa_aux: {
9931   \tex_ifnum:D \l_fp_input_a_integer_int > \c_one
9932     \tex_advance:D \l_fp_count_int \c_one
9933     \fp_ln_mantissa_divide_two:
9934     \tex_expandafter:D \fp_ln_mantissa_aux:
9935   \tex_fi:D
9936 }

```

A fast one-shot division by two.

```

9937 \cs_new_protected_nopar:Npn \fp_ln_mantissa_divide_two: {
9938   \tex_ifodd:D \l_fp_input_a_decimal_int
9939     \tex_advance:D \l_fp_input_a_extended_int \c_one_thousand_million
9940   \tex_fi:D
9941   \tex_ifodd:D \l_fp_input_a_integer_int
9942     \tex_advance:D \l_fp_input_a_decimal_int \c_one_thousand_million

```

```

9943 \tex_fi:D
9944 \tex_divide:D \l_fp_input_a_integer_int \c_two
9945 \tex_divide:D \l_fp_input_a_decimal_int \c_two
9946 \tex_divide:D \l_fp_input_a_extended_int \c_two
9947 }

```

Recovering constants makes use of the same auxiliary code as for exponents.

```

9948 \cs_new_protected_nopar:Npn \fp_ln_const:nn #1#2 {
9949   \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
9950     \fp_exp_integer_const:nnnn
9951     \tex_csname:D c_fp_ln_ #1 _ #2 _tl \tex_endcsname:D
9952 }

```

The Taylor series for the logarithm function is best implemented using the identity

$$\ln(x) = \ln\left(\frac{y+1}{y-1}\right)$$

with

$$y = \frac{x-1}{x+1}$$

This leads to the series

$$\ln(x) = 2y \left(1 + y^2 \left(\frac{1}{3} + y^2 \left(\frac{1}{5} + y^2 \left(\frac{1}{7} + y^2 \left(\frac{1}{9} + \cdots \right) \right) \right) \right) \right)$$

This expansion has the advantage that a lot of the work can be loaded up early by finding y^2 before the loop itself starts. (In practice, the implementation does the multiplication by two at the end of the loop, and expands out the brackets as this is an overall more efficient approach.)

At the implementation level, the code starts by calculating y and storing that in input **a** (which is no longer needed for other purposes). That is done using the full division system avoiding the parsing step. The value is then switched to a fixed-point representation. There is then some shuffling to get all of the working space set up. At this stage, a lot of registers are in use and so the Taylor series is calculated within a group so that the output variables can be used to hold the result. The value of y^2 is held in input **b** (there are a few assignments saved by choosing this over **a**), while input **a** is used for the ‘loop value’.

```

9953 \cs_new_protected_nopar:Npn \fp_ln_Taylor: {
9954   \group_begin:
9955     \l_fp_input_a_integer_int \c_zero
9956     \l_fp_input_a_exponent_int \c_zero
9957     \l_fp_input_b_integer_int \c_two
9958     \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
9959     \l_fp_input_b_exponent_int \c_zero
9960     \fp_div_internal:
9961     \fp_ln_fixed:

```

```

9962 \l_fp_input_a_integer_int \l_fp_output_integer_int
9963 \l_fp_input_a_decimal_int \l_fp_output_decimal_int
9964 \l_fp_input_a_exponent_int \l_fp_output_exponent_int
9965 \l_fp_input_a_extended_int \c_zero
9966 \l_fp_output_decimal_int \c_zero
9967 \l_fp_output_decimal_int \l_fp_input_a_decimal_int
9968 \l_fp_output_extended_int \l_fp_input_a_extended_int
9969 \fp_mul:NNNNNN
9970 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9971 \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
9972 \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
9973 \l_fp_count_int \c_one
9974 \fp_ln_Taylor_aux:
9975 \cs_set_protected_nopar:Npx \fp_tmp:w
9976 {
9977 \group_end:
9978 \exp_not:N \l_fp_exp_decimal_int
9979 \int_use:N \l_fp_output_decimal_int \scan_stop:
9980 \exp_not:N \l_fp_exp_extended_int
9981 \int_use:N \l_fp_output_extended_int \scan_stop:
9982 \exp_not:N \l_fp_exp_exponent_int
9983 \int_use:N \l_fp_output_exponent_int \scan_stop:
9984 }
9985 \fp_tmp:w

```

After the loop part of the Taylor series, the factor of 2 needs to be included. The total for the result can then be constructed.

```

9986 \tex_advance:D \l_fp_exp_decimal_int \l_fp_exp_decimal_int
9987 \tex_ifnum:D \l_fp_exp_extended_int < \c_five_hundred_million
9988 \tex_else:D
9989 \tex_advance:D \l_fp_exp_extended_int -\c_five_hundred_million
9990 \tex_advance:D \l_fp_exp_decimal_int \c_one
9991 \tex_fi:D
9992 \tex_advance:D \l_fp_exp_extended_int \l_fp_exp_extended_int
9993 \tex_ifnum:D \l_fp_output_sign_int > \c_zero
9994 \tex_expandafter:D \fp_add:NNNNNNNNN
9995 \tex_else:D
9996 \tex_expandafter:D \fp_sub:NNNNNNNNN
9997 \tex_fi:D
9998 \l_fp_output_integer_int \l_fp_output_decimal_int
9999 \l_fp_output_extended_int
10000 \c_zero \l_fp_exp_decimal_int \l_fp_exp_extended_int
10001 \l_fp_output_integer_int \l_fp_output_decimal_int
10002 \l_fp_output_extended_int
10003 }

```

The usual shifts to move to fixed-point working. This is done using the output registers as this saves a reassignment here.

```

10004 \cs_new_protected_nopar:Npn \fp_ln_fixed: {
10005   \tex_ifnum:D \l_fp_output_exponent_int < \c_zero
10006   \tex_advance:D \l_fp_output_decimal_int \c_one_thousand_million
10007   \tex_expandafter:D \use_i:nn \tex_expandafter:D
10008     \fp_ln_fixed_aux:NNNNNNNNN
10009     \int_use:N \l_fp_output_decimal_int
10010   \tex_expandafter:D \fp_ln_fixed:
10011   \tex_fi:D
10012 }
10013 \cs_new_protected_nopar:Npn
10014   \fp_ln_fixed_aux:NNNNNNNNN #1#2#3#4#5#6#7#8#9 {
10015   \tex_ifnum:D \l_fp_output_integer_int = \c_zero
10016     \l_fp_output_decimal_int #1#2#3#4#5#6#7#8 \scan_stop:
10017   \tex_else:D
10018     \tl_set:Nx \l_fp_tmp_tl
10019     {
10020       \int_use:N \l_fp_output_integer_int
10021       #1#2#3#4#5#6#7#8
10022     }
10023     \l_fp_output_integer_int \c_zero
10024     \l_fp_output_decimal_int \l_fp_tmp_tl \scan_stop:
10025   \tex_fi:D
10026   \tex_advance:D \l_fp_output_exponent_int \c_one
10027 }

```

The main loop for the Taylor series: unlike some of the other similar functions, the result here is not the final value and is therefore subject to further manipulation outside of the loop.

```

10028 \cs_new_protected_nopar:Npn \fp_ln_Taylor_aux: {
10029   \tex_advance:D \l_fp_count_int \c_two
10030   \fp_mul:NNNNNN
10031     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10032     \l_fp_input_b_decimal_int \l_fp_input_b_extended_int
10033     \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10034   \tex_ifnum:D
10035     \etex_numexpr:D
10036       \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
10037       > \c_zero
10038     \fp_div_integer:NNNNN
10039       \l_fp_input_a_decimal_int \l_fp_input_a_extended_int
10040       \l_fp_count_int
10041       \l_fp_exp_decimal_int \l_fp_exp_extended_int
10042     \tex_advance:D \l_fp_output_decimal_int \l_fp_exp_decimal_int
10043     \tex_advance:D \l_fp_output_extended_int \l_fp_exp_extended_int
10044     \tex_ifnum:D \l_fp_output_extended_int < \c_one_thousand_million
10045     \tex_else:D
10046       \tex_advance:D \l_fp_output_decimal_int \c_one
10047       \tex_advance:D \l_fp_output_extended_int
10048         -\c_one_thousand_million

```

```

10049 \tex_fi:D
10050 \tex_ifnum:D \l_fp_output_decimal_int < \c_one_thousand_million
10051 \tex_else:D
10052 \tex_advance:D \l_fp_output_integer_int \c_one
10053 \tex_advance:D \l_fp_output_decimal_int
10054 -\c_one_thousand_million
10055 \tex_fi:D
10056 \tex_expandafter:D \fp_ln_Taylor_aux:
10057 \tex_fi:D
10058 }

```

(End definition for `\fp_ln:Nn` and `\fp_ln:cn`. These functions are documented on page 154.)

`\fp_pow:Nn` The approach used for working out powers is to first filter out the various special cases and
`\fp_pow:cn` then do most of the work using the logarithm and exponent functions. The two storage
`\fp_gpow:Nn` areas are used in the reverse of the ‘natural’ logic as this avoids some re-assignment in
`\fp_gpow:cn` the sanity checking code.

```

\fp_pow_aux:NNn
\fp_pow_aux_i:
\fp_pow_positive:
\fp_pow_negative:
\fp_pow_aux_ii:
\fp_pow_aux_iii:
\fp_pow_aux_iv:
10059 \cs_new_protected_nopar:Npn \fp_pow:Nn {
10060 \fp_pow_aux:NNn \tl_set:Nn
10061 }
10062 \cs_new_protected_nopar:Npn \fp_gpow:Nn {
10063 \fp_pow_aux:NNn \tl_gset:Nn
10064 }
10065 \cs_generate_variant:Nn \fp_pow:Nn { c }
10066 \cs_generate_variant:Nn \fp_gpow:Nn { c }
10067 \cs_new_protected_nopar:Npn \fp_pow_aux:NNn #1#2#3 {
10068 \group_begin:
10069 \fp_read:N #2
10070 \l_fp_input_b_sign_int \l_fp_input_a_sign_int
10071 \l_fp_input_b_integer_int \l_fp_input_a_integer_int
10072 \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
10073 \l_fp_input_b_exponent_int \l_fp_input_a_exponent_int
10074 \fp_split:Nn a {#3}
10075 \fp_standardise:NNNN
10076 \l_fp_input_a_sign_int
10077 \l_fp_input_a_integer_int
10078 \l_fp_input_a_decimal_int
10079 \l_fp_input_a_exponent_int
10080 \tex_ifnum:D
10081 \etex_numexpr:D
10082 \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10083 = \c_zero
10084 \tex_ifnum:D
10085 \etex_numexpr:D
10086 \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10087 = \c_zero
10088 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10089 {
10090 \group_end:

```

```

10091         ##1 ##2 { \c_undefined_fp }
10092     }
10093     \tex_else:D
10094     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10095     {
10096         \group_end:
10097         ##1 ##2 { \c_zero_fp }
10098     }
10099     \tex_fi:D
10100 \tex_else:D
10101 \tex_ifnum:D
10102 \etex_numexpr:D
10103 \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10104 = \c_zero
10105 \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10106 {
10107     \group_end:
10108     ##1 ##2 { \c_one_fp }
10109 }
10110 \tex_else:D
10111 \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10112 \fp_pow_aux_i:
10113 \tex_fi:D
10114 \tex_fi:D
10115 \fp_tmp:w #1 #2
10116 }

```

Simply using the logarithm function directly will fail when negative numbers are raised to integer powers, which is a mathematically valid operation. So there are some more tests to make, after forcing the power into an integer and decimal parts, if necessary.

```

10117 \cs_new_protected_nopar:Npn \fp_pow_aux_i: {
10118     \tex_ifnum:D \l_fp_input_b_sign_int > \c_zero
10119     \tl_set:Nn \l_fp_sign_tl { + }
10120     \tex_expandafter:D \fp_pow_aux_ii:
10121 \tex_else:D
10122     \l_fp_input_a_extended_int \c_zero
10123     \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
10124     \group_begin:
10125     \fp_extended_normalise:
10126     \tex_ifnum:D
10127     \etex_numexpr:D
10128     \l_fp_input_a_decimal_int + \l_fp_input_a_extended_int
10129     = \c_zero
10130     \group_end:
10131     \tl_set:Nn \l_fp_sign_tl { - }
10132     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10133     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10134     \tex_expandafter:D \fp_pow_aux_ii:
10135 \tex_else:D

```

```

10136     \group_end:
10137     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10138     {
10139         \group_end:
10140         ##1 ##2 { \c_undefined_fp }
10141     }
10142     \tex_fi:D
10143 \tex_else:D
10144     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10145     {
10146         \group_end:
10147         ##1 ##2 { \c_undefined_fp }
10148     }
10149     \tex_fi:D
10150 \tex_fi:D
10151 }

```

The approach used here for powers works well in most cases but gives poorer results for negative integer powers, which often have exact values. So there is some filtering to do. For negative powers where the power is small, an alternative approach is used in which the positive value is worked out and the reciprocal is then taken. The filtering is unfortunately rather long.

```

10152 \cs_new_protected_nopar:Npn \fp_pow_aux_ii: {
10153     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
10154     \tex_expandafter:D \fp_pow_aux_iv:
10155 \tex_else:D
10156     \tex_ifnum:D \l_fp_input_a_exponent_int < \c_ten
10157     \group_begin:
10158     \l_fp_input_a_extended_int \c_zero
10159     \fp_extended_normalise:
10160     \tex_ifnum:D \l_fp_input_a_decimal_int = \c_zero
10161     \tex_ifnum:D \l_fp_input_a_integer_int > \c_ten
10162     \group_end:
10163     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10164     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10165     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10166     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10167     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10168     \fp_pow_aux_iv:
10169 \tex_else:D
10170     \group_end:
10171     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10172     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10173     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10174     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10175     \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10176     \tex_expandafter:D \fp_pow_aux_iii:
10177     \tex_fi:D
10178 \tex_else:D

```

```

10179         \group_end:
10180         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10181         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10182         \tex_expandafter:D \fp_pow_aux_iv:
10183         \tex_fi:D
10184     \tex_else:D
10185         \tex_expandafter:D \tex_expandafter:D \tex_expandafter:D
10186         \fp_pow_aux_iv:
10187         \tex_fi:D
10188     \tex_fi:D
10189     \cs_set_protected_nopar:Npx \fp_tmp:w ##1##2
10190     {
10191         \group_end:
10192         ##1 ##2
10193         {
10194             \l_fp_sign_tl
10195             \int_use:N \l_fp_output_integer_int
10196             .
10197             \tex_expandafter:D \use_none:n
10198             \tex_number:D \etex_numexpr:D
10199             \l_fp_output_decimal_int + \c_one_thousand_million
10200             \scan_stop:
10201             e
10202             \int_use:N \l_fp_output_exponent_int
10203         }
10204     }
10205 }

```

For the small negative integer powers, the calculation is done for the positive power and the reciprocal is then taken.

```

10206 \cs_new_protected_nopar:Npn \fp_pow_aux_iii: {
10207     \l_fp_input_a_sign_int \c_one
10208     \fp_pow_aux_iv:
10209     \l_fp_input_a_integer_int \c_one
10210     \l_fp_input_a_decimal_int \c_zero
10211     \l_fp_input_a_exponent_int \c_zero
10212     \l_fp_input_b_integer_int \l_fp_output_integer_int
10213     \l_fp_input_b_decimal_int \l_fp_output_decimal_int
10214     \l_fp_input_b_exponent_int \l_fp_output_exponent_int
10215     \fp_div_internal:
10216 }

```

The business end of the code starts by finding the logarithm of the given base. There is a bit of a shuffle so that this does not have to be re-parsed and so that the output ends up in the correct place. There is also a need to enable using the short-cut for a pre-calculated result. The internal part of the multiplication function can then be used to do the second part of the calculation directly. There is some more set up before doing the exponential: the idea here is to deactivate some internals so that everything works

smoothly.

```

10217 \cs_new_protected_nopar:Npn \fp_pow_aux_iv: {
10218   \group_begin:
10219     \l_fp_input_a_integer_int \l_fp_input_b_integer_int
10220     \l_fp_input_a_decimal_int \l_fp_input_b_decimal_int
10221     \l_fp_input_a_exponent_int \l_fp_input_b_exponent_int
10222     \fp_ln_internal:
10223     \cs_set_protected_nopar:Npx \fp_tmp:w
10224     {
10225       \group_end:
10226       \exp_not:N \l_fp_input_b_sign_int
10227       \int_use:N \l_fp_output_sign_int \scan_stop:
10228       \exp_not:N \l_fp_input_b_integer_int
10229       \int_use:N \l_fp_output_integer_int \scan_stop:
10230       \exp_not:N \l_fp_input_b_decimal_int
10231       \int_use:N \l_fp_output_decimal_int \scan_stop:
10232       \exp_not:N \l_fp_input_b_extended_int
10233       \int_use:N \l_fp_output_extended_int \scan_stop:
10234       \exp_not:N \l_fp_input_b_exponent_int
10235       \int_use:N \l_fp_output_exponent_int \scan_stop:
10236     }
10237     \fp_tmp:w
10238     \l_fp_input_a_extended_int \c_zero
10239     \fp_mul:NNNNNNNNN
10240     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
10241     \l_fp_input_a_extended_int
10242     \l_fp_input_b_integer_int \l_fp_input_b_decimal_int
10243     \l_fp_input_b_extended_int
10244     \l_fp_input_a_integer_int \l_fp_input_a_decimal_int
10245     \l_fp_input_a_extended_int
10246     \tex_advance:D \l_fp_input_a_exponent_int \l_fp_input_b_exponent_int
10247     \l_fp_output_integer_int \c_zero
10248     \l_fp_output_decimal_int \c_zero
10249     \l_fp_output_exponent_int \c_zero
10250     \cs_set_eq:NN \fp_exp_const:Nx \use_none:nn
10251     \fp_exp_internal:
10252   }

```

(End definition for `\fp_pow:Nn` and `\fp_pow:cn`. These functions are documented on page 154.)

118.13 Tests for special values

`\fp_if_undefined_p:N` Testing for an undefined value is easy.
`\fp_if_undefined:NTF`

```

10253 \prg_new_conditional:Npnn \fp_if_undefined:N #1 { T , F , TF , p } {
10254   \tex_ifx:D #1 \c_undefined_fp
10255   \prg_return_true:
10256   \tex_else:D

```

```

10257     \prg_return_false:
10258     \tex_fi:D
10259 }

```

(End definition for `\fp_if_undefined_p:N`. This function is documented on page 151.)

`\fp_if_zero_p:N` Testing for a zero fixed-point is also easy.
`\fp_if_zero:NTF`

```

10260 \prg_new_conditional:Npnn \fp_if_zero:N #1 { T , F , TF , p } {
10261   \tex_ifx:D #1 \c_zero_fp
10262   \prg_return_true:
10263   \tex_else:D
10264   \prg_return_false:
10265   \tex_fi:D
10266 }

```

(End definition for `\fp_if_zero_p:N`. This function is documented on page 151.)

118.14 Floating-point conditionals

`\fp_compare:nNnTF`

`\fp_compare:NNNTF`

`\fp_compare_aux:N`

`\fp_compare_=:`

`\fp_compare_<:`

`\fp_compare_<_aux:`

`\fp_compare_absolute_a>b:`

`\fp_compare_absolute_a<b:`

`\fp_compare_>:`

The idea for the comparisons is to provide two versions: slower and faster. The lead off for both is the same: get the two numbers read and then look for a function to handle the comparison.

```

10267 \prg_new_protected_conditional:Npnn \fp_compare:nNn #1#2#3
10268 { T , F , TF }
10269 {
10270   \group_begin:
10271     \fp_split:Nn a {#1}
10272     \fp_standardise:NNNN
10273     \l_fp_input_a_sign_int
10274     \l_fp_input_a_integer_int
10275     \l_fp_input_a_decimal_int
10276     \l_fp_input_a_exponent_int
10277     \fp_split:Nn b {#3}
10278     \fp_standardise:NNNN
10279     \l_fp_input_b_sign_int
10280     \l_fp_input_b_integer_int
10281     \l_fp_input_b_decimal_int
10282     \l_fp_input_b_exponent_int
10283     \fp_compare_aux:N #2
10284   }
10285 \prg_new_protected_conditional:Npnn \fp_compare:NNN #1#2#3
10286 { T , F , TF }
10287 {
10288   \group_begin:
10289     \fp_read:N #3
10290     \l_fp_input_b_sign_int      \l_fp_input_a_sign_int
10291     \l_fp_input_b_integer_int   \l_fp_input_a_integer_int

```

```

10292 \l_fp_input_b_decimal_int \l_fp_input_a_decimal_int
10293 \l_fp_input_b_exponent_int \l_fp_input_a_exponent_int
10294 \fp_read:N #1
10295 \fp_compare_aux:N #2
10296 }
10297 \cs_new_protected_nopar:Npn \fp_compare_aux:N #1 {
10298 \cs_if_exist:cTF { fp_compare_#1: }
10299 { \use:c { fp_compare_#1: } }
10300 {
10301 \group_end:
10302 \prg_return_false:
10303 }
10304 }

```

For equality, the test is pretty easy as things are either equal or they are not.

```

10305 \cs_new_protected_nopar:cpn { fp_compare_=: } {
10306 \tex_ifnum:D \l_fp_input_a_sign_int = \l_fp_input_b_sign_int
10307 \tex_ifnum:D \l_fp_input_a_integer_int = \l_fp_input_b_integer_int
10308 \tex_ifnum:D \l_fp_input_a_decimal_int = \l_fp_input_b_decimal_int
10309 \tex_ifnum:D
10310 \l_fp_input_a_exponent_int = \l_fp_input_b_exponent_int
10311 \group_end:
10312 \prg_return_true:
10313 \tex_else:D
10314 \group_end:
10315 \prg_return_false:
10316 \tex_fi:D
10317 \tex_else:D
10318 \group_end:
10319 \prg_return_false:
10320 \tex_fi:D
10321 \tex_else:D
10322 \group_end:
10323 \prg_return_false:
10324 \tex_fi:D
10325 \tex_else:D
10326 \group_end:
10327 \prg_return_false:
10328 \tex_fi:D
10329 }

```

Comparing two values is quite complex. First, there is a filter step to check if one or other of the given values is zero. If it is then the result is relatively easy to determine.

```

10330 \cs_new_protected_nopar:cpn { fp_compare_>: } {
10331 \tex_ifnum:D \etex_numexpr:D
10332 \l_fp_input_a_integer_int + \l_fp_input_a_decimal_int
10333 = \c_zero
10334 \tex_ifnum:D \etex_numexpr:D
10335 \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int

```

```

10336     = \c_zero
10337     \group_end:
10338     \prg_return_false:
10339   \tex_else:D
10340     \tex_ifnum:D \l_fp_input_b_sign_int > \c_zero
10341     \group_end:
10342     \prg_return_false:
10343     \tex_else:D
10344     \group_end:
10345     \prg_return_true:
10346     \tex_fi:D
10347   \tex_fi:D
10348 \tex_else:D
10349   \tex_ifnum:D \etex_numexpr:D
10350     \l_fp_input_b_integer_int + \l_fp_input_b_decimal_int
10351     = \c_zero
10352     \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
10353     \group_end:
10354     \prg_return_true:
10355     \tex_else:D
10356     \group_end:
10357     \prg_return_false:
10358     \tex_fi:D
10359   \tex_else:D
10360     \use:c { fp_compare_>_aux: }
10361   \tex_fi:D
10362 \tex_fi:D
10363 }

```

Next, check the sign of the input: this again may give an obvious result. If both signs are the same, then hand off to comparing the absolute values.

```

10364 \cs_new_protected_nopar:cpn { fp_compare_>_aux: } {
10365   \tex_ifnum:D \l_fp_input_a_sign_int > \l_fp_input_b_sign_int
10366   \group_end:
10367   \prg_return_true:
10368 \tex_else:D
10369   \tex_ifnum:D \l_fp_input_a_sign_int < \l_fp_input_b_sign_int
10370   \group_end:
10371   \prg_return_false:
10372 \tex_else:D
10373   \tex_ifnum:D \l_fp_input_a_sign_int > \c_zero
10374   \use:c { fp_compare_absolute_a>b: }
10375 \tex_else:D
10376   \use:c { fp_compare_absolute_a<b: }
10377   \tex_fi:D
10378 \tex_fi:D
10379 \tex_fi:D
10380 }

```

Rather long runs of checks, as there is the need to go through each layer of the input and do the comparison. There is also the need to avoid messing up with equal inputs at each stage.

```

10381 \cs_new_protected_nopar:cpn { fp_compare_absolute_a>b: } {
10382   \tex_ifnum:D \l_fp_input_a_exponent_int > \l_fp_input_b_exponent_int
10383   \group_end:
10384   \prg_return_true:
10385 \tex_else:D
10386   \tex_ifnum:D \l_fp_input_a_exponent_int < \l_fp_input_b_exponent_int
10387   \group_end:
10388   \prg_return_false:
10389 \tex_else:D
10390   \tex_ifnum:D \l_fp_input_a_integer_int > \l_fp_input_b_integer_int
10391   \group_end:
10392   \prg_return_true:
10393 \tex_else:D
10394   \tex_ifnum:D
10395   \l_fp_input_a_integer_int < \l_fp_input_b_integer_int
10396   \group_end:
10397   \prg_return_false:
10398 \tex_else:D
10399   \tex_ifnum:D
10400   \l_fp_input_a_decimal_int > \l_fp_input_b_decimal_int
10401   \group_end:
10402   \prg_return_true:
10403 \tex_else:D
10404   \group_end:
10405   \prg_return_false:
10406   \tex_fi:D
10407   \tex_fi:D
10408   \tex_fi:D
10409   \tex_fi:D
10410 \tex_fi:D
10411 }
10412 \cs_new_protected_nopar:cpn { fp_compare_absolute_a<b: } {
10413   \tex_ifnum:D \l_fp_input_b_exponent_int > \l_fp_input_a_exponent_int
10414   \group_end:
10415   \prg_return_true:
10416 \tex_else:D
10417   \tex_ifnum:D \l_fp_input_b_exponent_int < \l_fp_input_a_exponent_int
10418   \group_end:
10419   \prg_return_false:
10420 \tex_else:D
10421   \tex_ifnum:D \l_fp_input_b_integer_int > \l_fp_input_a_integer_int
10422   \group_end:
10423   \prg_return_true:
10424 \tex_else:D
10425   \tex_ifnum:D
10426   \l_fp_input_b_integer_int < \l_fp_input_a_integer_int

```

```

10427         \group_end:
10428         \prg_return_false:
10429     \tex_else:D
10430         \tex_ifnum:D
10431             \l_fp_input_b_decimal_int > \l_fp_input_a_decimal_int
10432         \group_end:
10433         \prg_return_true:
10434     \tex_else:D
10435         \group_end:
10436         \prg_return_false:
10437     \tex_fi:D
10438 \tex_fi:D
10439 \tex_fi:D
10440 \tex_fi:D
10441 \tex_fi:D
10442 }

```

This is just a case of reversing the two input values and then running the tests already defined.

```

10443 \cs_new_protected_nopar:cpn { fp_compare_<: } {
10444     \tl_set:Nx \l_fp_tmp_tl
10445     {
10446         \int_set:Nn \exp_not:N \l_fp_input_a_sign_int
10447         { \int_use:N \l_fp_input_b_sign_int }
10448         \int_set:Nn \exp_not:N \l_fp_input_a_integer_int
10449         { \int_use:N \l_fp_input_b_integer_int }
10450         \int_set:Nn \exp_not:N \l_fp_input_a_decimal_int
10451         { \int_use:N \l_fp_input_b_decimal_int }
10452         \int_set:Nn \exp_not:N \l_fp_input_a_exponent_int
10453         { \int_use:N \l_fp_input_b_exponent_int }
10454         \int_set:Nn \exp_not:N \l_fp_input_b_sign_int
10455         { \int_use:N \l_fp_input_a_sign_int }
10456         \int_set:Nn \exp_not:N \l_fp_input_b_integer_int
10457         { \int_use:N \l_fp_input_a_integer_int }
10458         \int_set:Nn \exp_not:N \l_fp_input_b_decimal_int
10459         { \int_use:N \l_fp_input_a_decimal_int }
10460         \int_set:Nn \exp_not:N \l_fp_input_b_exponent_int
10461         { \int_use:N \l_fp_input_a_exponent_int }
10462     }
10463     \l_fp_tmp_tl
10464     \use:c { fp_compare_>: }
10465 }

```

(End definition for `\fp_compare:nNn`. This function is documented on page ??.)

118.15 Messages

`\fp_overflow_msg:` A generic overflow message, used whenever there is a possible overflow.

```

10466 \msg_kernel_new:nnnn { fpu } { overflow }
10467 { Number~too~big. }
10468 {
10469   The~input~given~is~too~big~for~the~LaTeX~floating~point~unit. \\
10470   Further~errors~may~well~occur!
10471 }
10472 \cs_new_protected_nopar:Npn \fp_overflow_msg: {
10473   \msg_kernel_error:nn { fpu } { overflow }
10474 }

```

(End definition for \fp_overflow_msg:. This function is documented on page ??.)

\fp_exp_overflow_msg: A slightly more helpful message for exponent overflows.

```

10475 \msg_kernel_new:nnnn { fpu } { exponent-overflow }
10476 { Number~too~big~for~exponent~unit. }
10477 {
10478   The~exponent~of~the~input~given~is~too~big~for~the~floating~point~
10479   unit:~the~maximum~input~value~for~an~exponent~is~230.
10480 }
10481 \cs_new_protected_nopar:Npn \fp_exp_overflow_msg: {
10482   \msg_kernel_error:nn { fpu } { exponent-overflow }
10483 }

```

(End definition for \fp_exp_overflow_msg:. This function is documented on page ??.)

\fp_ln_error_msg: Logarithms are only valid for positive number

```

10484 \msg_kernel_new:nnnn { fpu } { logarithm-input-error }
10485 { Invalid~input~to~ln~function. }
10486 { Logarithms~can~only~be~calculated~for~positive~numbers. }
10487 \cs_new_protected_nopar:Npn \fp_ln_error_msg: {
10488   \msg_kernel_error:nn { fpu } { logarithm-input-error }
10489 }

```

(End definition for \fp_ln_error_msg:. This function is documented on page ??.)

\fp_trig_overflow_msg: A slightly more helpful message for trigonometric overflows.

```

10490 \msg_kernel_new:nnnn { fpu } { trigonometric-overflow }
10491 { Number~too~big~for~trigonometry~unit. }
10492 {
10493   The~trigonometry~code~can~only~work~with~numbers~smaller~
10494   than~1000000000.
10495 }
10496 \cs_new_protected_nopar:Npn \fp_trig_overflow_msg: {
10497   \msg_kernel_error:nn { fpu } { trigonometric-overflow }
10498 }

```

(End definition for \fp_trig_overflow_msg:. This function is documented on page ??.)

```

10499 </initex | package>

```

119 Implementation

Announce and ensure that the required packages are loaded.

```
10500 <*package>
10501 \ProvidesExplPackage
10502   {\filename}{\filedate}{\fileversion}{\filedescription}
10503 \package_check_loaded_expl:
10504 </package>
10505 <*initex | package>
```

```
\lua_now:n
\lua_now:x
\lua_shipout_x:n
\lua_shipout_x:x
\lua_shipout:n
\lua_shipout:x
\lua_wrong_engine:

10506 \luatex_if_engine:TF
10507 {
10508   \cs_new_eq:NN \lua_now:x \luatex_directlua:D
10509   \cs_new_eq:NN \lua_shipout_x:n \luatex_latelua:D
10510 }
10511 {
10512   \cs_new:Npn \lua_now:x #1 { \lua_wrong_engine: }
10513   \cs_new_protected:Npn \lua_shipout_x:n #1 { \lua_wrong_engine: }
10514 }
10515 \cs_new:Npn \lua_now:n #1 {
10516   \lua_now:x { \exp_not:n {#1} }
10517 }
10518 \cs_generate_variant:Nn \lua_shipout_x:n { x }
10519 \cs_new_protected:Npn \lua_shipout:n #1 {
10520   \lua_shipout_x:n { \exp_not:n {#1} }
10521 }
10522 \cs_generate_variant:Nn \lua_shipout:n { x }
10523 \group_begin:
10524 \char_make_letter:N\!
10525 \char_make_letter:N\%
10526 \cs_gset:Npn\lua_wrong_engine:{%
10527 \LuaTeX engine not in use!%
10528 }%
10529 \group_end:%
```

(End definition for `\lua_now:n`. This function is documented on page ??.)

119.1 Category code tables

```
\g_cctab_allocate_int
\g_cctab_stack_int
\g_cctab_stack_seq
```

To allocate category code tables, both the read-only and stack tables need to be followed. There is also a sequence stack for the dynamic tables themselves.


```

10530 \int_new:N \g_cctab_allocate_int
10531 \int_set:Nn \g_cctab_allocate_int { -1 }
10532 \int_new:N \g_cctab_stack_int
10533 \seq_new:N \g_cctab_stack_seq

```

(End definition for `\g_cctab_allocate_int`. This function is documented on page ??.)

`\cctab_new:N` Creating a new category code table is done slightly differently from other registers. Low-numbered tables are more efficiently-stored than high-numbered ones. There is also a need to have a stack of flexible tables as well as the set of read-only ones. To satisfy both of these requirements, odd numbered tables are used for read-only tables, and even ones for the stack. Here, therefore, the odd numbers are allocated.

```

10534 \cs_new_protected_nopar:Npn \cctab_new:N #1 {
10535   \cs_if_free:NTF #1
10536   {
10537     \int_gadd:Nn \g_cctab_allocate_int { 2 }
10538     \int_compare:nNnTF
10539     { \g_cctab_allocate_int } < { \c_allocate_max_tl + 1 }
10540     {
10541       \tex_global:D \tex_mathchardef:D #1 \g_cctab_allocate_int
10542       \luatex_initcatcodetable:D #1
10543     }
10544     {
10545       \msg_kernel_error:nxx { code } { out-of-registers } { cctab }
10546     }
10547   }
10548   {
10549     \msg_kernel_error:nxx { code } { variable-already-defined }
10550     { \token_to_str:N #1 }
10551   }
10552 }
10553 \luatex_if_engine:F {
10554   \cs_set_protected_nopar:Npn \cctab_new:N #1 { \lua_wrong_engine: }
10555 }
10556 <*package>
10557 \luatex_if_engine:T {
10558   \cs_set_protected_nopar:Npn \cctab_new:N #1
10559   {
10560     \newcatcodetable #1
10561     \luatex_initcatcodetable:D #1
10562   }
10563 }
10564 </package>

```

(End definition for `\cctab_new:N`. This function is documented on page 157.)

`\cctab_begin:N` The aim here is to ensure that the saved tables are read-only. This is done by using a
`\cctab_end:` stack of tables which are not read only, and actually having them as 'in use' copies.
`\l_cctab_tmp_tl`

```

10565 \cs_new_protected_nopar:Npn \cctab_begin:N #1 {
10566   \seq_gpush:Nx \g_cctab_stack_seq { \tex_the:D \luatex_catcodetable:D }
10567   \luatex_catcodetable:D #1
10568   \int_gadd:Nn \g_cctab_stack_int { 2 }
10569   \int_compare:nNnT { \g_cctab_stack_int } > { 268435453 }
10570     { \msg_kernel_error:nn { code } { cctab-stack-full } }
10571   \luatex_savecatcodetable:D \g_cctab_stack_int
10572   \luatex_catcodetable:D \g_cctab_stack_int
10573 }
10574 \cs_new_protected_nopar:Npn \cctab_end: {
10575   \int_gsub:Nn \g_cctab_stack_int { 2 }
10576   \seq_gpop:NN \g_cctab_stack_seq \l_cctab_tmp_tl
10577   \quark_if_no_value:NT \l_cctab_tmp_tl
10578     { \tl_set:Nn \l_cctab_tmp_tl { 0 } }
10579   \luatex_catcodetable:D \l_cctab_tmp_tl \scan_stop:
10580 }
10581 \luatex_if_engine:F {
10582   \cs_set_protected_nopar:Npn \cctab_begin:N #1 { \lua_wrong_engine: }
10583   \cs_set_protected_nopar:Npn \cctab_end: { \lua_wrong_engine: }
10584 }
10585 <*package>
10586 \luatex_if_engine:T {
10587   \cs_set_protected_nopar:Npn \cctab_begin:N #1
10588     { \BeginCatcodeRegime #1 }
10589   \cs_set_protected_nopar:Npn \cctab_end:
10590     { \EndCatcodeRegime }
10591 }
10592 </package>
10593 \tl_new:N \l_cctab_tmp_tl

```

(End definition for \cctab_begin:N. This function is documented on page ??.)

\cctab_gset:Nn Category code tables are always global, so only one version is needed. The set up here is simple, and means that at the point of use there is no need to worry about escaping category codes.

```

10594 \cs_new_protected:Npn \cctab_gset:Nn #1#2 {
10595   \group_begin:
10596     #2
10597     \luatex_savecatcodetable:D #1
10598   \group_end:
10599 }
10600 \luatex_if_engine:F {
10601   \cs_set_protected_nopar:Npn \cctab_gset:Nn #1#2 { \lua_wrong_engine: }
10602 }

```

(End definition for \cctab_gset:Nn. This function is documented on page 157.)

\c_code_cctab Creating category code tables is easy using the function above. The other and string
\c_document_cctab ones are done by completely ignoring the existing codes as this makes life a lot less
\c_initex_cctab
\c_other_cctab
\c_string_cctab

complex. The table for expl3 category codes is always needed, whereas when in package mode the rest can be copied from the existing L^AT_EX 2_ε package `luatex`.

```

10603 \luatex_if_engine:T {
10604   \cctab_new:N \c_code_cctab
10605   \cctab_gset:Nn \c_code_cctab { }
10606 }
10607 <*package>
10608 \luatex_if_engine:T {
10609   \cs_new_eq:NN \c_document_cctab \CatcodeTableLaTeX
10610   \cs_new_eq:NN \c_initex_cctab \CatcodeTableIniTeX
10611   \cs_new_eq:NN \c_other_cctab \CatcodeTableOther
10612   \cs_new_eq:NN \c_string_cctab \CatcodeTableString
10613 }
10614 </package>
10615 <!*package>
10616 \luatex_if_engine:T {
10617   \cctab_new:N \c_document_cctab
10618   \cctab_new:N \c_other_cctab
10619   \cctab_new:N \c_string_cctab
10620   \cctab_gset:Nn \c_document_cctab
10621     {
10622       \char_make_space:n { 9 }
10623       \char_make_space:n { 32 }
10624       \char_make_other:n { 58 }
10625       \char_make_subscript:n { 95 }
10626       \char_make_active:n { 126 }
10627     }
10628   \cctab_gset:Nn \c_other_cctab
10629     {
10630       \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
10631       { \char_make_other:n {#1} }
10632     }
10633   \cctab_gset:Nn \c_string_cctab
10634     {
10635       \prg_stepwise_inline:nnnn { 0 } { 1 } { 127 }
10636       { \char_make_other:n {#1} }
10637       \char_make_space:n { 32 }
10638     }
10639   }
10640 </!package>

```

(End definition for `\c_code_cctab`. This function is documented on page 158.)

```

10641 </initex | package>

```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
\!	1640, 5930, 10524
\#	4
\.	2305, 2306, 2318, 2319, 5928, 5930, 5931
*	2615, 2617, 2621, 2626
\,	6393, 6395
\-	145
\.	5877, 5932
\.bool_gset:N	6835
\.bool_set:N	6835
\.choice:	6841
\.choice_code:n	6844
\.choice_code:x	6844
\.code:n	6850
\.code:x	6850
\.default:V	6856
\.default:n	6856
\.dim_gset:N	6862
\.dim_gset:c	6862
\.dim_set:N	6862
\.dim_set:c	6862
\.fp_gset:N	6874
\.fp_gset:c	6874
\.fp_set:N	6874
\.fp_set:c	6874
\.generate_choices:n	6886
\.int_gset:N	6889
\.int_gset:c	6889
\.int_set:N	6889
\.int_set:c	6889
\.meta:n	6901
\.meta:x	6901
\.skip_gset:N	6907
\.skip_gset:c	6907
\.skip_set:N	6907
\.skip_set:c	6907
\.tl_gset:N	6919
\.tl_gset:c	6919
\.tl_gset_x:N	6919
\.tl_gset_x:c	6919
\.tl_set:N	6919
\.tl_set:c	6919
\.tl_set_x:N	6919
\.tl_set_x:c	6919
\.value_forbidden:	6943
\.value_required:	6943
\/	144
\:	684, 688, 1142, 2796
\::	210, 1582, 1585, 1587, 1587–1589, 1591, 1592, 1594, 1597, 1600, 1606, 1610, 1618, 1661–1668, 1670–1677, 1679–1681, 1683, 1685– 1687, 1689–1692, 1694–1697, 1699– 1704, 1706, 1707, 1800, 1804, 1807, 1811, 1817–1820, 1822, 1825, 1831
\::N	210, 1591, 1591, 1665–1668, 1687, 1689–1692, 1699, 1700, 1825, 1831
\::V	210, 1610, 1610, 1663, 1667, 1676, 1683, 1687
\::V_unbraced	1799, 1807, 1818, 1822, 1825, 1831
\::c	210, 1594, 1594, 1670, 1673, 1679, 1694, 1699–1704, 1822
\::f	210, 1600, 1600, 1661, 1665, 1671–1674
\::f_unbraced	1799, 1800, 1817
\::n	210, 1588, 1588, 1674– 1677, 1689, 1690, 1694–1697, 1704
\::o	210, 1597, 1597, 1671, 1675, 1679–1681, 1685, 1689, 1691, 1692, 1695, 1697, 1700–1702, 1706, 1707
\::o_unbraced	1799, 1804, 1819
\::v	210, 1610, 1618, 1662, 1666
\::v_unbraced	1799, 1811, 1820
\::x	210, 1606, 1606, 1664, 1668, 1670, 1677, 1681, 1685, 1686, 1690, 1692, 1696, 1697, 1703, 1704, 1706
\=	6392, 6394
\?	1758, 2709, 2796
\@	818, 1142, 1143, 4410, 4411
\@end	787
\@hyph	791
\@input	785
\@italiccorr	790
\@underline	786
\@addtofilelist	7087
\@currentx	817

\@currname 816, 7019
 \@currnamestack 811, 819
 \@declaredoptions 814
 \@empty 813, 814
 \@filelist 7112
 \@gobble 797
 \@ifpackageloaded 831
 \@l@expl@log@functions@bool 1264
 \@nil 811, 815
 \@p@pfilename 811, 815
 \@popfilename 159, 760, 767, 768
 \@pushfilename 159, 760, 760, 765
 \@tempa 45, 46, 54
 \@tempboxa 5492
 \@unknownoptionerror 812
 \\ 1250,
 1271, 1758, 2709, 5778, 5785, 5862,
 5871, 5873, 5937, 5943, 5948, 5950,
 5959, 5964, 5972, 5979, 6213, 6220,
 6228, 6545, 6952, 6959, 6967, 6968,
 6974, 6981, 6988, 6994, 7000, 10469
 \{ 2, 5064, 5155, 5295, 5297
 \} 3, 5064, 5155, 5295, 5297, 5929
 ^ 5, 8, 78
 _ 683, 687, 2318
 _cs_generate_variant_aux:Ncpx
 1756, 1768, 1774
 _cs_generate_variant_aux:ccpx
 1734, 1756, 1773
 _cs_generate_variant_aux:w
 1756, 1770, 1775
 _l_tl_replace_tl .. 4482, 4482, 4491,
 4493, 4495, 4517, 4523, 4527, 4537
 _tl_if_single_aux:w ... 4317, 4327, 4330
 _tl_replace_all_in_aux:NNnn
 4510, 4511, 4514, 4516
 _tl_replace_in_aux:NNnn
 4482, 4484, 4486, 4506
 _tl_tmp:w 4487, 4492, 4494,
 4500, 4518, 4522, 4524, 4528, 4533
 \l 3271

Numbers

\8 6394
 \9 6395
 _ 143, 1134, 1641, 5812, 5928,
 5929, 5938, 5973, 5980, 6281, 10525

A

\A 2710, 2798, 4410, 4412
 \above 264
 \abovedisplayshortskip 277
 \abovedisplayskip 278
 \abovewithdelims 265
 \accent 315
 \adjdemerits 352
 \advance 159
 \afterassignment 169
 \aftergroup 170
 \alloc_new:nnnN
 3045, 3791, 3911, 4033, 4601, 5412
 \alloc_reg:NnnN 5603, 5607
 \alloc_setup_type:nnn 5601, 5605
 \AtBeginDocument 6311, 7110
 \atop 266
 \atopwithdelims 267

B

\B 4411, 4413
 \badness 414
 \baselineskip 342
 \batchmode 235
 \begin 6326
 \BeginCatcodeRegime 10588
 \begingroup 11, 51, 57, 61, 173
 \beginL 527
 \beginR 529
 \belowdisplayshortskip 279
 \belowdisplayskip 280
 \binoppenalty 303
 \bool(:w 2045
 \bool_)_0:w 2045
 \bool_)_1:w 2045
 \bool_8_0:w 2045
 \bool_8_1:w 2045
 \bool:w 2045
 \bool_choose:NN 2045, 2124, 2131
 \bool_cleanup:N .. 2045, 2116, 2121, 2123
 \bool_do_until:cn 36, 2037
 \bool_do_until:Nn 36, 2037, 2041, 2042, 2044
 \bool_do_until:nn .. 39, 2173, 2182, 2183
 \bool_do_while:cn 36, 2037
 \bool_do_while:Nn 36, 2037, 2037, 2038, 2040
 \bool_do_while:nn .. 39, 2173, 2179, 2180
 \bool_eval_skip_to_end:Nw
 2143–2146, 2147, 2147, 2156
 \bool_eval_skip_to_end_aux:Nw
 2147, 2148, 2150

\bool_eval_skip_to_end_auxii:Nw	\bool_set:Nn
2147, 2153, 2155	37, 2158, 2158, 2162
\bool_get_next:N	\bool_set_eq:cc
2045, 2047, 2049,	35, 2012, 2015
2064, 2097, 2116, 2119, 2133–2136	\bool_set_eq:cN
\bool_get_next:NN	35, 2012, 2014
2063, 2074	\bool_set_eq:Nc
\bool_get_not_next:N	35, 2012, 2013
2056, 2067, 2094	\bool_set_eq:NN
\bool_get_not_next:NN	35, 2012, 2012
2066, 2086	\bool_set_false:c
\bool_gset:cn	35, 2002, 2007
37, 2158	\bool_set_false:N
.bool_gset:N	35, 2002, 2006, 6662, 6778, 7960, 7987
137	\bool_set_true:c
\bool_gset:Nn	35, 2002, 2005
37, 2158, 2159, 2163	\bool_set_true:N
\bool_gset_eq:cc	35, 2002, 2004, 6364, 6658, 6774, 7995
35, 2012, 2019	\bool_until_do:cn
\bool_gset_eq:cN	36, 2029
35, 2012, 2018	\bool_until_do:Nn 36, 2029, 2033, 2034, 2036
\bool_gset_eq:Nc	\bool_until_do:nn
35, 2012, 2017	39, 2173, 2176, 2177
\bool_gset_eq:NN	\bool_while_do:cn
35, 2012, 2016	36, 2029
\bool_gset_false:c	\bool_while_do:Nn 36, 2029, 2029, 2030, 2032
35, 2002, 2011	\bool_while_do:nn
\bool_gset_false:N	39, 2173, 2173, 2174
35, 2002, 2010	\bool_xor_p:nn
\bool_gset_true:c	37, 2165, 2165
35, 2002, 2009	\botmark
\bool_gset_true:N	250
35, 2002, 2008	\botmarks
\bool_I_0:w	476
2045	\box
\bool_I_1:w	458
2045	\box_clear:c
\bool_if:cTF	114, 5463
36, 2022	\box_clear:N
\bool_if:N	114, 5463, 5463–5465
2022	\box_dp:c
\bool_if:n	114, 5467
2169	\box_dp:N
\bool_if:NF	114, 5467, 5468, 5471, 5474
2028, 2034, 2042, 3463	\box_gc_clear:c
\bool_if:nF	114, 5463
2177, 2183	\box_gc_clear:N
\bool_if:NT	114, 5463, 5465, 5466
2027, 2030,	\box_gset_eq:cc
2038, 3463, 3464, 6427, 7969	113, 5450
\bool_if:nT	\box_gset_eq:cN
2174, 2180	113, 5450
\bool_if:NTF	\box_gset_eq:Nc
36, 1434, 2022, 2026,	113, 5450
3449, 6498, 6677, 6786, 6798, 7981	\box_gset_eq:NN
\bool_if:nTF	113, 5450, 5450, 5451
37, 2045, 2925, 3893	\box_gset_eq_clear:cc
\bool_if_p:c	113, 5450
36, 2022	\box_gset_eq_clear:cN
\bool_if_p:N	113, 5450
36, 2022, 2025	\box_gset_eq_clear:Nc
\bool_if_p:n	113, 5450
37, 2045,	\box_gset_eq_clear:NN 113, 5450, 5452, 5453
2045, 2158, 2160, 2164, 2166, 2170	\box_gset_to_last:c
\bool_new:c	114, 5455
35, 2002, 2003	\box_gset_to_last:N 114, 5455, 5457, 5458
\bool_new:N	114, 5467
35, 2002, 2002,	\box_ht:N
2020, 2021, 6363, 6566, 6580, 7188	114, 5467, 5467, 5470, 5477
\bool_Not:N	\box_if_empty:cTF
2076, 2099	113, 5439
\bool_Not:w	\box_if_empty:N
2045, 2071, 2093	5439
\bool_not_choose:NN	\box_if_empty:NF
2128, 2132	5445
\bool_not_cleanup:N	\box_if_empty:NT
2119, 2122, 2127	5444
\bool_not_Not:N	\box_if_empty:NTF
2088, 2107	113, 5439, 5443
\bool_not_Not:w	\box_if_empty_p:c
2083, 2096	113, 5439
\bool_not_p:n	\box_if_empty_p:N
37, 2164, 2164	113, 5439, 5442
\bool_p:w	\box_if_horizontal:cTF
2045, 2100, 2108	112, 5425
\bool_S_0:w	\box_if_horizontal:N
2045	5425
\bool_S_1:w	\box_if_horizontal:NF
2045	5434
\bool_set:cn	\box_if_horizontal:NT
37, 2158	5433
.bool_set:N	
137	

<code>\box_if_horizontal:NTF</code>	112 , 5425 , 5432
<code>\box_if_horizontal_p:c</code>	112 , 5425
<code>\box_if_horizontal_p:N</code>	112 , 5425 , 5431
<code>\box_if_vertical:cTF</code>	113 , 5425
<code>\box_if_vertical:N</code>	5428
<code>\box_if_vertical:NF</code>	5438
<code>\box_if_vertical:NT</code>	5437
<code>\box_if_vertical:NTF</code>	113 , 5425 , 5436
<code>\box_if_vertical_p:c</code>	113 , 5425
<code>\box_if_vertical_p:N</code>	113 , 5425 , 5435
<code>\box_move_down:nn</code>	114 , 5459 , 5462
<code>\box_move_left:nn</code>	114 , 5459 , 5459
<code>\box_move_right:nn</code>	114 , 5459 , 5460
<code>\box_move_up:nn</code>	114 , 5459 , 5461
<code>\box_new:c</code>	112 , 5411
<code>\box_new:N</code>	112 , 5411 , 5416 , 5421 , 5493 – 5495
<code>\box_set_dp:cn</code>	115 , 5473
<code>\box_set_dp:Nn</code>	115 , 5473 , 5473 , 5483
<code>\box_set_eq:cc</code>	113 , 5446
<code>\box_set_eq:cN</code>	113 , 5446
<code>\box_set_eq:Nc</code>	113 , 5446
<code>\box_set_eq:NN</code>	113 , 5446 , 5446 , 5447 , 5450
<code>\box_set_eq_clear:cc</code>	113 , 5448
<code>\box_set_eq_clear:cN</code>	113 , 5448
<code>\box_set_eq_clear:Nc</code>	113 , 5448
<code>\box_set_eq_clear:NN</code>	113 , 5448 , 5448 , 5449 , 5452 , 5463
<code>\box_set_ht:cn</code>	115 , 5473
<code>\box_set_ht:Nn</code>	115 , 5473 , 5476 , 5482
<code>\box_set_to_last:c</code>	114 , 5455
<code>\box_set_to_last:N</code>	114 , 5455 , 5455 – 5457
<code>\box_set_wd:cn</code>	115 , 5473
<code>\box_set_wd:Nn</code>	115 , 5473 , 5479 , 5484
<code>\box_show:c</code>	115 , 5489
<code>\box_show:N</code>	115 , 5489 , 5489 , 5490
<code>\box_use:c</code>	114 , 5485
<code>\box_use:N</code>	114 , 5485 , 5487 , 5488
<code>\box_use_clear:c</code>	114 , 5485
<code>\box_use_clear:N</code>	114 , 5485 , 5485 , 5486
<code>\box_wd:c</code>	114 , 5467
<code>\box_wd:N</code>	114 , 5467 , 5469 , 5472 , 5480
<code>\boxmaxdepth</code>	420
<code>\brokenpenalty</code>	377
C	
<code>\C</code>	1764 , 2710 , 2799
<code>\c_active_char_token</code>	49 , 2612 , 2627 , 2670
<code>\c_alignment_tab_token</code>	49 , 2612 , 2618 , 2642
<code>\c_allocate_max_tl</code>	10539
<code>\c_code_cctab</code>	158 , 10603 , 10604 , 10605
<code>\c_document_cctab</code>	158 , 10603 , 10609 , 10617 , 10620
<code>\c_e_fp</code>	147 , 7142 , 7142 , 7143
<code>\c_eight</code>	66 , 2539 , 2555 , 3691 , 3701 , 5577 , 7961
<code>\c_eleven</code>	66 , 2542 , 2558 , 3691 , 3704 , 5580
<code>\c_empty_box</code>	115 , 5463 , 5491 , 5491 , 5493
<code>\c_empty_tl</code>	83 , 4132 , 4134 , 4245 , 4245 , 4256 , 4807 , 5100
<code>\c_empty_toks</code>	90 , 4652 , 4744 , 4744
<code>\c_false_bool</code>	11 , 1059 , 1088 , 1130 , 1131 , 1156 , 1413 , 1420 , 2002 , 2003 , 2006 , 2007 , 2010 , 2011 , 2102 , 2112 , 2137 , 2140 , 2141 , 2143 , 2145 , 2167 , 2752 , 2846 , 3440 , 3444 , 3452 , 4580
<code>\c_fifteen</code>	66 , 2546 , 2562 , 3691 , 3708 , 5584
<code>\c_five</code>	66 , 2536 , 2552 , 3691 , 3698 , 5574 , 7615 , 8853 , 9152
<code>\c_five_hundred_million</code>	7123 , 7128 , 7661 , 9616 , 9798 , 9987 , 9989
<code>\c_forty_four</code>	7123 , 7123 , 7778
<code>\c_four</code>	66 , 2535 , 2551 , 3691 , 3697 , 5573 , 7759 , 7994
<code>\c_fourteen</code>	66 , 2545 , 2561 , 3691 , 3707 , 5583
<code>\c_fp_exp_100_tl</code>	9335
<code>\c_fp_exp_10_tl</code>	9335
<code>\c_fp_exp_1_tl</code>	9335
<code>\c_fp_exp_200_tl</code>	9335
<code>\c_fp_exp_20_tl</code>	9335
<code>\c_fp_exp_2_tl</code>	9335
<code>\c_fp_exp_30_tl</code>	9335
<code>\c_fp_exp_3_tl</code>	9335
<code>\c_fp_exp_40_tl</code>	9335
<code>\c_fp_exp_4_tl</code>	9335
<code>\c_fp_exp_50_tl</code>	9335
<code>\c_fp_exp_5_tl</code>	9335
<code>\c_fp_exp_60_tl</code>	9335
<code>\c_fp_exp_6_tl</code>	9335
<code>\c_fp_exp_70_tl</code>	9335
<code>\c_fp_exp_7_tl</code>	9335
<code>\c_fp_exp_80_tl</code>	9335
<code>\c_fp_exp_8_tl</code>	9335
<code>\c_fp_exp_90_tl</code>	9335
<code>\c_fp_exp_9_tl</code>	9335
<code>\c_fp_exp_100_tl</code>	9275
<code>\c_fp_exp_10_tl</code>	9275
<code>\c_fp_exp_1_tl</code>	9275
<code>\c_fp_exp_200_tl</code>	9275
<code>\c_fp_exp_20_tl</code>	9275
<code>\c_fp_exp_2_tl</code>	9275

<code>\c_fp_exp_30_tl</code>	9275	<code>\c_int_from_roman_m_int</code>	3722
<code>\c_fp_exp_3_tl</code>	9275	<code>\c_int_from_roman_V_int</code>	3722
<code>\c_fp_exp_40_tl</code>	9275	<code>\c_int_from_roman_v_int</code>	3722
<code>\c_fp_exp_4_tl</code>	9275	<code>\c_int_from_roman_X_int</code>	3722
<code>\c_fp_exp_50_tl</code>	9275	<code>\c_int_from_roman_x_int</code>	3722
<code>\c_fp_exp_5_tl</code>	9275	<code>\c_io_streams_tl</code>	122
<code>\c_fp_exp_60_tl</code>	9275	<code>\c_ior_log_stream</code> .	123 , 5563 , 5566 , 5622
<code>\c_fp_exp_6_tl</code>	9275	<code>\c_ior_streams_tl</code>	5567 , 5586 , 5641
<code>\c_fp_exp_70_tl</code>	9275	<code>\c_ior_term_stream</code>	123 , 5563 , 5564
<code>\c_fp_exp_7_tl</code>	9275	<code>\c_iow_log_stream</code>
<code>\c_fp_exp_80_tl</code>	9275	...	123 , 5563 , 5565 , 5618 , 5793 , 5794
<code>\c_fp_exp_8_tl</code>	9275	<code>\c_iow_streams_tl</code>	5567 , 5567 , 5586 , 5628
<code>\c_fp_exp_90_tl</code>	9275	<code>\c_iow_term_stream</code>
<code>\c_fp_exp_9_tl</code>	9275	...	123 , 5563 , 5563 , 5795 , 5796
<code>\c_fp_ln_10_1_tl</code>	9695	<code>\c_job_name_tl</code>	83 , 4243 , 4243 , 4244
<code>\c_fp_ln_10_2_tl</code>	9695	<code>\c_keys_properties_root_tl</code>
<code>\c_fp_ln_10_3_tl</code>	9695	...	144 , 6555 , 6556 , 6668 , 6754 , 6757
<code>\c_fp_ln_10_4_tl</code>	9695	<code>\c_keys_root_tl</code>	144 , 6555 ,
<code>\c_fp_ln_10_5_tl</code>	9695	...	6555 , 6585 , 6605 , 6608 , 6627 , 6632 ,
<code>\c_fp_ln_10_6_tl</code>	9695	...	6637 – 6639 , 6642 – 6644 , 6703 , 6711 ,
<code>\c_fp_ln_10_7_tl</code>	9695	...	6719 , 6760 , 6809 , 6815 , 6817 , 6823
<code>\c_fp_ln_10_8_tl</code>	9695	<code>\c_keys_value_forbidden_tl</code>
<code>\c_fp_ln_10_9_tl</code>	9695	...	145 , 6557 , 6557
<code>\c_fp_ln_2_1_tl</code>	9722	<code>\c_keys_value_required_tl</code>	145 , 6557 , 6558
<code>\c_fp_ln_2_2_tl</code>	9722	<code>\c_KV_single_equal_sign_tl</code>
<code>\c_fp_ln_2_3_tl</code>	9722	...	134 , 6356 , 6358 , 6469 , 6520
<code>\c_fp_pi_by_four_decimal_int</code> .	7130 ,	<code>\c_letter_token</code>	49 , 2612 , 2624 , 2662
...	7130 , 7131 , 8789 , 8801 , 8808 , 8812	<code>\c_luatex_is_engine_bool</code>
<code>\c_fp_pi_by_four_extended_int</code>	25 , 1411 , 1419 , 1420 , 1426
...	7130 , 7132 , 7133 , 8789 , 8801 , 8813	<code>\c_math_shift_token</code> .	49 , 2612 , 2616 , 2638
<code>\c_fp_pi_decimal_int</code>	7130 , 7134 , 7135 , 8731	<code>\c_math_subscript_token</code>
<code>\c_fp_pi_extended_int</code> ..	7130 , 7136 , 7137	...	49 , 2612 , 2622 , 2654
<code>\c_fp_two_pi_decimal_int</code>	<code>\c_math_superscript_token</code>
...	7130 , 7138 , 7139 , 8727 , 8733	...	49 , 2612 , 2620 , 2650
<code>\c_fp_two_pi_extended_int</code>	<code>\c_max_dim</code>	75 , 3966 , 3967
...	7130 , 7140 , 7141 , 8727 , 8733	<code>\c_max_int</code>	66 , 3721 , 3721
<code>\c_group_begin_token</code>	<code>\c_max_register_int</code>
...	49 , 2612 , 2612 , 2630 , 2926 , 5513 , 5539	...	66 , 1232 , 1232 , 3045 ,
<code>\c_group_end_token</code>	49 , 2612 , 2613 ,	...	3674 , 3791 , 3911 , 4033 , 4601 , 5412
...	2634 , 2927 , 5514 , 5517 , 5542 , 5547	<code>\c_max_skip</code> ..	71 , 3882 , 3885 , 3886 , 3890 , 3967
<code>\c_hundred_one</code>	66 , 3691 , 3711	<code>\c_minus_one</code>	16 , 66 , 1220 ,
<code>\c_initex_cctab</code>	158 , 10603 , 10610	...	1221 , 1224 , 1225 , 1234 , 3084 , 3102 ,
<code>\c_int_from_roman_C_int</code>	3722	...	3104 , 3672 , 3691 , 3691 , 3692 , 4421 ,
<code>\c_int_from_roman_c_int</code>	3722	...	4437 , 4447 , 4455 , 5565 , 5566 , 5748 ,
<code>\c_int_from_roman_D_int</code>	3722	...	5760 , 5841 , 7205 , 7311 , 7745 , 7999 ,
<code>\c_int_from_roman_d_int</code>	3722	...	8000 , 8144 , 8182 , 8426 , 8472 , 8476 ,
<code>\c_int_from_roman_I_int</code>	3722	...	8634 , 8749 , 8753 , 9010 , 9024 , 9110 ,
<code>\c_int_from_roman_i_int</code>	3722	...	9114 , 9188 , 9196 , 9679 , 9683 , 9833
<code>\c_int_from_roman_L_int</code>	3722	<code>\c_msg_coding_error_text_tl</code>
<code>\c_int_from_roman_M_int</code>	3722	...	130 , 5848 , 5848 , 6212 , 6965

- \c_msg_error_tl [130](#), [5842](#), 5842, 6085, 6183
- \c_msg_fatal_text_tl [130](#), [5848](#), 5852, 6080, 6177
- \c_msg_help_text_tl [130](#), [5848](#), 5855, 5950
- \c_msg_hide_tl 5880, 5951
- \c_msg_hide_tl<spaces> [5876](#)
- \c_msg_info_tl [130](#), [5842](#), 5844, 6109
- \c_msg_kernel_bug_more_text_tl
..... [130](#), [5848](#), 5861, 6238
- \c_msg_kernel_bug_text_tl
..... [130](#), [5848](#), 5858, 6234
- \c_msg_more_text_prefix_tl
..... [130](#), [5884](#), 5885,
5915, 5920, 6092, 6094, 6192, 6194
- \c_msg_no_info_text_tl
[130](#), [5848](#), 5866, 5940, 5961, 6097, 6197
- \c_msg_on_line_tl . [130](#), [5883](#), 5883, 5898
- \c_msg_return_text_tl [130](#), [5848](#),
5864, 5868, 5870, 6215, 6223, 6230
- \c_msg_text_prefix_tl
..... [130](#), [5884](#), 5884, 5905,
5909, 5914, 5919, 6036, 6076, 6087,
6103, 6110, 6116, 6122, 6172, 6185
- \c_msg_warning_tl . [130](#), [5842](#), 5843, 6102
- \c_nine [66](#), 2540,
2556, [3691](#), 3702, 5578, 7316, 8668,
8884, 8972, 9215, 9224, 9523, 9837
- \c_one [66](#),
2532, 2548, 3079, 3101, 3103, [3691](#),
3694, 5570, 7207, 7217, 7277, 7288,
7335, 7341, 7380, 7403, 7613, 7970,
7974, 7976, 7982, 8130, 8158, 8422,
8458, 8463, 8483, 8603, 8665, 8704,
8717, 8766, 8780, 8804, 8816, 8879,
8967, 9015, 9021, 9036, 9061, 9082,
9087, 9095, 9101, 9190, 9194, 9477,
9486, 9582, 9607, 9618, 9622, 9644,
9664, 9670, 9800, 9804, 9835, 9848,
9903, 9925, 9931, 9932, 9973, 9990,
10026, 10046, 10052, 10207, 10209
- \c_one_fp ... [147](#), [7144](#), 7144, 7145, 10108
- \c_one_hundred [7123](#), 7124, 7338, 7339, 9485
- \c_one_hundred_million
..... [7123](#), 7127, 8345, 9246
- \c_one_million [7123](#), 7126, 8601
- \c_one_thousand
.. [7123](#), 7125, 8254, 8506, 8549, 8599
- \c_one_thousand_million
... [7123](#), 7129, 7280, 7300, 7317,
7327, 7361, 7371, 7384, 7394, 7445,
7488, 7762, 7781, 7901, 7938, 7963,
8015, 8043, 8112, 8128, 8131, 8145,
8154, 8237, 8274, 8282, 8289, 8380,
8392, 8427, 8456, 8459, 8461, 8464,
8473, 8477, 8484, 8485, 8602, 8604,
8614, 8625, 8638, 8669, 8679, 8695,
8750, 8754, 8770, 8774, 8849, 8903,
8948, 8991, 9041, 9047, 9093, 9097,
9099, 9103, 9111, 9115, 9148, 9269,
9423, 9590, 9600, 9619, 9637, 9662,
9666, 9668, 9672, 9680, 9684, 9781,
9801, 9824, 9876, 9939, 9942, 10006,
10044, 10048, 10050, 10054, 10199
- \c_other_cctab
.... [158](#), [10603](#), 10611, 10618, 10628
- \c_other_char_token . [49](#), [2612](#), 2625, 2666
- \c_parameter_token [49](#), [2612](#), 2619
- \c_peek_true_remove_next_tl
..... [247](#), 2895, [2906](#), 2907
- \c_pi_fp [147](#), [7146](#), 7146, 7147
- \c_seven [66](#), [1220](#), 1230,
1884, 2538, 2554, [3691](#), 3700, 5576
- \c_six [66](#), [1220](#), 1229, 1880, 2537,
2553, [3691](#), 3699, 5575, 8727, 8733
- \c_sixteen [16](#),
[66](#), [1220](#), 1227, 1237, [3691](#), 3709,
5563, 5564, 5601, 5605, 5627, 5629,
5640, 5642, 5672, 5693, 5712, 5733
- \c_space_tl [83](#), 1250, 1271,
[4246](#), 4246, 5046, 5052, 5063, 5155,
5295–5297, 5899, 5938, 5973, 5980,
6102, 6105, 6109, 6112, 6183, 6966
- \c_space_token
.... [49](#), [2612](#), 2623, 2658, 2928, 3019
- \c_string_cctab
.... [158](#), [10603](#), 10612, 10619, 10633
- \c_ten [66](#), 2541,
2557, [3691](#), 3703, 5579, 6281, 7328,
7372, 7395, 7556, 7623, 7687, 7780,
7786, 7898, 7935, 7972, 7975, 7984,
8391, 8442, 8613, 8658, 8696, 8707,
8782, 9176, 9896, 10123, 10156, 10161
- \c_ten_thousand [66](#), [3691](#), 3715
- \c_ten_thousand_four [66](#), [3691](#), 3719
- \c_ten_thousand_one [66](#), [3691](#), 3716
- \c_ten_thousand_three ... [66](#), [3691](#), 3718
- \c_ten_thousand_two [66](#), [3691](#), 3717
- \c_thirteen [66](#), 2544, 2560, [3691](#), 3706, 5582
- \c_thirty_two [66](#), [3691](#), 3710
- \c_thousand [66](#), [3691](#), 3714

- `\c_three` 66, 2534, 2550,
3691, 3696, 5572, 8725, 9033, 9447
- `\c_true_bool` 11, 1059, 1088,
1130, 1130, 1160, 1412, 1419, 2004,
2005, 2008, 2009, 2101, 2104, 2109,
2110, 2138, 2139, 2142, 2144, 2146,
2167, 3436, 3440, 3444, 3456, 4578
- `\c_twelve` 66, 1220, 1231, 2306,
2319, 2543, 2559, 3691, 3705, 5581
- `\c_twenty_thousand` 66, 3691, 3720
- `\c_two` 66, 2533, 2549, 3691, 3695, 5571,
7748, 8730, 9009, 9013, 9031, 9064,
9187, 9193, 9944–9946, 9957, 10029
- `\c_two_ats_with_two_catcodes_tl`
. 4409, 4416, 4420, 4431, 4454
- `\c_twohundred_fifty_five` . 66, 3691, 3712
- `\c_twohundred_fifty_six` . 66, 3691, 3713
- `\c_undefined` 126, 132, 5920, 6760
- `\c_undefined:D` . . . 1390, 1393, 1396, 1399
- `\c_undefined_fp` 147, 7148, 7148, 7149,
8322, 9203, 10091, 10140, 10147, 10254
- `\c_xetex_is_engine_bool`
. 25, 1411, 1412, 1413, 1415, 1422
- `\c_zero` 16, 66, 1058,
1066, 1074, 1081, 1087, 1095, 1103,
1110, 1202, 1208, 1220, 1228, 1290,
1294, 1877, 1878, 1882, 1931, 1957,
1981, 2489, 2499, 2531, 2547, 3109,
3111, 3580, 3610, 3614, 3615, 3621,
3691, 3693, 3791, 3894, 3895, 3911,
4033, 4601, 5412, 5569, 5601, 5605,
6680, 7225, 7236, 7276, 7287, 7289,
7299, 7342–7344, 7451, 7494, 7536,
7539, 7605, 7679, 7803–7811, 7841–
7849, 7907, 7944, 7985, 8046, 8087,
8103, 8143, 8147, 8149, 8221, 8225,
8249, 8318, 8328, 8342, 8343, 8364,
8368, 8388, 8398, 8399, 8425, 8471,
8475, 8479, 8480, 8498, 8541, 8612,
8637, 8647, 8655, 8708, 8711, 8718–
8720, 8748, 8752, 8756, 8761, 8783,
8784, 8789, 8797, 8801, 8812, 8840,
8880, 8894, 8939, 8968, 8982, 9008,
9020, 9022, 9034, 9035, 9037, 9038,
9040, 9042, 9045, 9055–9057, 9088,
9089, 9109, 9113, 9139, 9186, 9200,
9201, 9230, 9231, 9243, 9244, 9260,
9411, 9414, 9449, 9474, 9478–9480,
9487–9489, 9501, 9533, 9550, 9551,
9580, 9581, 9586, 9587, 9592, 9621,
9657, 9658, 9678, 9682, 9747, 9751,
9803, 9815, 9831, 9854–9857, 9862,
9885, 9893, 9906, 9907, 9909, 9912,
9955, 9956, 9959, 9965, 9966, 9993,
10000, 10005, 10015, 10023, 10037,
10083, 10087, 10104, 10118, 10122,
10129, 10153, 10158, 10160, 10210,
10211, 10238, 10247–10249, 10333,
10336, 10340, 10351, 10352, 10373
- `\c_zero_dim` 75, 3966, 3966, 5519
- `\c_zero_fp` 147, 7150, 7150, 7151, 7408,
7424, 7427, 8332, 9180, 9233, 9440,
9466, 9504, 9758, 9767, 10097, 10261
- `\c_zero_skip` 71, 3816, 3882, 3883, 3884,
3889, 3901, 3902, 3942, 3966, 5549
- `\catcode` 2–5, 8, 100–
102, 104–109, 112–114, 116–121, 462
- `\catcodetable` 636
- `\CatcodeTableIniTeX` 10610
- `\CatcodeTableLaTeX` 10609
- `\CatcodeTableOther` 10611
- `\CatcodeTableString` 10612
- `\cctab_begin:N`
. 158, 10565, 10565, 10582, 10587
- `\cctab_end:` 158, 10565, 10574, 10583, 10589
- `\cctab_gset:Nn` 157, 10594, 10594,
10601, 10605, 10620, 10628, 10633
- `\cctab_new:N` 157, 10534, 10534,
10554, 10558, 10604, 10617–10619
- `\char` 316
- `\char_gset_mathcode:nn` . . 49, 2563, 2568
- `\char_gset_mathcode:w` 49, 2563, 2567, 2569
- `\char_make_active:N` . 47, 2531, 2544, 5931
- `\char_make_active:n` 47, 2547, 2560, 10626
- `\char_make_alignment:N` . . 47, 2531, 2535
- `\char_make_alignment:n` . . 47, 2547, 2551
- `\char_make_begin_group:N` . 47, 2531, 2532
- `\char_make_begin_group:n` . 47, 2547, 2548
- `\char_make_comment:N` 47, 2531, 2545
- `\char_make_comment:n` 47, 2547, 2561
- `\char_make_end_group:N` . . 47, 2531, 2533
- `\char_make_end_group:n` . . 47, 2547, 2549
- `\char_make_end_line:N` . . . 47, 2531, 2536
- `\char_make_end_line:n` . . . 47, 2547, 2552
- `\char_make_escape:N` 47, 2531, 2531
- `\char_make_escape:n` 47, 2547, 2547
- `\char_make_ignore:N` 47, 2531, 2540
- `\char_make_ignore:n` . 47, 2547, 2556, 7215
- `\char_make_invalid:N` 47, 2531, 2546
- `\char_make_invalid:n` 47, 2547, 2562

- \char_make_letter:N 47,
2531, 2542, 5877, 5932, 10524, 10525
- \char_make_letter:n 47, 2547, 2558
- \char_make_math_shift:N . 47, 2531, 2534
- \char_make_math_shift:n . 47, 2547, 2550
- \char_make_math_subscript:N
..... 47, 2531, 2539
- \char_make_math_subscript:n
..... 47, 2547, 2555
- \char_make_math_superscript:N
..... 47, 2531, 2538
- \char_make_math_superscript:n
..... 47, 2547, 2554
- \char_make_other:N 47, 2531, 2543
- \char_make_other:n
.. 47, 2547, 2559, 10624, 10631, 10636
- \char_make_parameter:N .. 47, 2531, 2537
- \char_make_parameter:n .. 47, 2547, 2553
- \char_make_space:N 47, 2531, 2541
- \char_make_space:n
.. 47, 2547, 2557, 10622, 10623, 10637
- \char_make_subscript:n 10625
- \char_set_catcode:nn
..... 46, 2517, 2518, 2531–
2562, 2615, 2617, 2621, 2626, 2711,
2797–2801, 6281, 6392, 6393, 6405
- \char_set_catcode:w . 46, 2517, 2517, 2519
- \char_set_lccode:nn .. 48, 2579, 2580,
2704–2709, 2796, 5928, 6394, 6395
- \char_set_lccode:w
.... 48, 2579, 2579, 2581, 5929, 5930
- \char_set_mathcode:nn ... 49, 2563, 2564
- \char_set_mathcode:w 49, 2563, 2563, 2565
- \char_set_sfcode:nn 48, 2599, 2600
- \char_set_sfcode:w .. 48, 2599, 2599, 2601
- \char_set_uccode:nn 48, 2589, 2590
- \char_set_uccode:w .. 48, 2589, 2589, 2591
- \char_show_value_catcode:n 46, 2517, 2528
- \char_show_value_catcode:w
..... 46, 2517, 2525, 2529
- \char_show_value_lccode:n 48, 2579, 2587
- \char_show_value_lccode:w
..... 48, 2579, 2586, 2588
- \char_show_value_mathcode:n
..... 49, 2563, 2576
- \char_show_value_mathcode:w
..... 49, 2563, 2575, 2577
- \char_show_value_sfcode:n 48, 2599, 2608
- \char_show_value_sfcode:w
..... 48, 2599, 2607, 2609
- \char_show_value_uccode:n 48, 2589, 2597
- \char_show_value_uccode:w
..... 48, 2589, 2596, 2598
- \char_value_catcode:n ... 46, 2517, 2522
- \char_value_catcode:w 46, 2517, 2521, 2523
- \char_value_lccode:n 48, 2579, 2584
- \char_value_lccode:w 48, 2579, 2583, 2584
- \char_value_mathcode:n .. 49, 2563, 2572
- \char_value_mathcode:w 49, 2563, 2571, 2573
- \char_value_sfcode:n 48, 2599, 2604
- \char_value_sfcode:w 48, 2599, 2603, 2605
- \char_value_uccode:n 48, 2589, 2594
- \char_value_uccode:w 48, 2589, 2593, 2594
- \chardef 151
- \chk_exist_cs:N 4118, 4122
- \chk_global:N 4123, 4630, 4640
- \chk_if_exist_cs:c 11, 1277, 1284
- \chk_if_exist_cs:N
..... 11, 1277, 1277, 1285, 1720
- \chk_if_free_cs:c
.... 11, 1246, 1260, 5905, 5909, 5997
- \chk_if_free_cs:N
..... 11, 1246, 1246, 1261, 1265,
1303, 1375, 3049, 3680, 3795, 3875,
3915, 4037, 4057, 4065, 4605, 5417
- \chk_local:N 4613, 4635
- \chk_local_or_pref_global:N .. 3056,
3081, 3086, 3116, 3122, 3803, 3818,
3833, 3840, 4119, 4653, 4687, 4691
- \chk_var_or_const:N 4059, 4066,
4119, 4123, 4138, 4150, 4636, 4641
- .choice: 137
- .choice_code:n 137
- .choice_code:x 137
- \cleaders 334
- \clearpage 6330, 6333, 6338, 6341
- \clist_clear:c 101, 5081
- \clist_clear:N
101, 5081, 5081, 5082, 5216, 5238, 6021
- \clist_clear_new:c 101, 5085
- \clist_clear_new:N . 101, 5085, 5085, 5086
- \clist_concat:ccc 105, 5201
- \clist_concat:NNN . 105, 5201, 5210, 5212
- \clist_concat_aux:NNNN
..... 5201, 5201, 5210, 5211
- \clist_display:c 103, 5146
- \clist_display:N .. 103, 5146, 5146, 5160
- \clist_gclear:c 101, 5081
- \clist_gclear:N ... 101, 5081, 5083, 5084
- \clist_gclear_new:c 101, 5085

<code>\clist_gclear_new:N</code>	101 , 5085 , 5087 , 5088	<code>\clist_if_eq:NNTF</code>	105 , 5105
<code>\clist_gconcat:ccc</code>	105 , 5201	<code>\clist_if_eq_p:cc</code>	105 , 5105
<code>\clist_gconcat:NNN</code>	105 , 5201 , 5211 , 5213	<code>\clist_if_eq_p:cN</code>	105 , 5105
<code>\clist_get:cN</code>	103 , 5128 , 5261	<code>\clist_if_eq_p:Nc</code>	105 , 5105
<code>\clist_get:NN</code>	103 , 5128 , 5128 , 5132 , 5260	<code>\clist_if_eq_p:NN</code>	105 , 5105
<code>\clist_get_aux:w</code>	107 , 5130 , 5133 , 5133	<code>\clist_if_in:cnTF</code>	105 , 5109
<code>\clist_gpop:cN</code>	107 , 5254	<code>\clist_if_in:coTF</code>	105 , 5109
<code>\clist_gpop:NN</code>	107 , 5254 , 5258 , 5259	<code>\clist_if_in:cVTF</code>	105 , 5109
<code>\clist_gpush:cn</code>	106 , 5254	<code>\clist_if_in:Nn</code>	5109
<code>\clist_gpush:Nn</code>	106 , 5254 , 5254 , 5257	<code>\clist_if_in:NnF</code>	5118 , 5221 , 5923
<code>\clist_gpush:No</code>	106 , 5254 , 5256	<code>\clist_if_in:NnT</code>	5117
<code>\clist_gpush:NV</code>	106 , 5254 , 5255	<code>\clist_if_in:NnTF</code>	105 , 5109 , 5116 , 6025
<code>\clist_gput_left:cn</code>	102 , 5170	<code>\clist_if_in:NoTF</code>	105 , 5109
<code>\clist_gput_left:co</code>	102 , 5170	<code>\clist_if_in:NVTF</code>	105 , 5109
<code>\clist_gput_left:cV</code>	102 , 5170	<code>\clist_map_break:</code>	104 , 5182
<code>\clist_gput_left:Nn</code>	102 , 5170 , 5170 , 5173 , 5254	<code>\clist_map_function:cc</code>	103 , 5182
<code>\clist_gput_left:No</code>	102 , 5170 , 5256	<code>\clist_map_function:cN</code>	103 , 5182
<code>\clist_gput_left:NV</code>	102 , 5170 , 5255	<code>\clist_map_function:Nc</code>	103 , 5182
<code>\clist_gput_left:Nx</code>	102 , 5170	<code>\clist_map_function:nc</code>	103 , 5182
<code>\clist_gput_right:cn</code>	102 , 5178	<code>\clist_map_function:NN</code>	103 , 5182 , 5183–5185 , 5217 , 5244
<code>\clist_gput_right:co</code>	102 , 5178	<code>\clist_map_function:nN</code>	103 , 5182 , 6614
<code>\clist_gput_right:cV</code>	102 , 5178	<code>\clist_map_inline:cn</code>	103
<code>\clist_gput_right:Nn</code>	102 , 5178 , 5178 , 5181	<code>\clist_map_inline:Nn</code>	103 , 5150 , 5182 , 5186 , 5187 , 7052 , 7112
<code>\clist_gput_right:No</code>	102 , 5178	<code>\clist_map_inline:nN</code>	103 , 5182
<code>\clist_gput_right:NV</code>	102 , 5178	<code>\clist_map_variable:cNn</code>	104 , 5188
<code>\clist_gput_right:Nx</code>	102 , 5178	<code>\clist_map_variable:NNn</code>	104 , 5188 , 5194 , 5195
<code>\clist_gremove_duplicates:N</code>	106 , 5215 , 5228	<code>\clist_map_variable:nNn</code>	104 , 5188 , 5188 , 5194
<code>\clist_gremove_element:Nn</code>	106 , 5231 , 5234	<code>\clist_map_variable_aux:Nnw</code>	5190 , 5196 , 5196 , 5199
<code>\clist_gset_eq:cc</code>	100 , 5093 , 5096	<code>\clist_new:c</code>	100 , 5079
<code>\clist_gset_eq:cN</code>	100 , 5093 , 5094	<code>\clist_new:N</code>	100 , 5079 , 5079 , 5080 , 5214 , 5889 , 5892
<code>\clist_gset_eq:Nc</code>	100 , 5093 , 5095	<code>\clist_pop:cN</code>	107 , 5248
<code>\clist_gset_eq:NN</code>	100 , 5093 , 5093 , 5229 , 5235	<code>\clist_pop:NN</code>	107 , 5248 , 5252 , 5253
<code>\clist_if_empty:c</code>	5098	<code>\clist_pop_aux:nnNN</code>	107 , 5134 , 5134 , 5252 , 5258
<code>\clist_if_empty:cTF</code>	104 , 5097	<code>\clist_pop_aux:w</code>	107 , 5134 , 5136 , 5138
<code>\clist_if_empty:N</code>	5097	<code>\clist_pop_auxi:w</code>	107 , 5134 , 5141 , 5143
<code>\clist_if_empty:NNTF</code>	104 , 5097 , 5162	<code>\clist_push:cn</code>	106 , 5248 , 5251
<code>\clist_if_empty_err:N</code>	107 , 5099 , 5099 , 5129 , 5135	<code>\clist_push:Nn</code>	106 , 5248 , 5248
<code>\clist_if_empty_p:c</code>	104 , 5097	<code>\clist_push:No</code>	106 , 5248 , 5250
<code>\clist_if_empty_p:N</code>	104 , 5097	<code>\clist_push:NV</code>	106 , 5248 , 5249
<code>\clist_if_eq:cc</code>	5108	<code>\clist_put_aux:NNnnNn</code>	107 , 5161 , 5161 , 5167 , 5171 , 5175 , 5179
<code>\clist_if_eq:ccTF</code>	105 , 5105	<code>\clist_put_left:cn</code>	101 , 5166 , 5251
<code>\clist_if_eq:cN</code>	5106	<code>\clist_put_left:co</code>	101 , 5166
<code>\clist_if_eq:cNTF</code>	105 , 5105		
<code>\clist_if_eq:Nc</code>	5107		
<code>\clist_if_eq:NcTF</code>	105 , 5105		
<code>\clist_if_eq:NN</code>	5105		

\clist_put_left:cV	101, 5166	1653, 1655, 1657–1659, 1718, 1890,
\clist_put_left:Nn		1896, 1899, 2646, 3142, 4242, 6276
.....	101 , 5166 , 5166, 5169, 5248	\cs_end:
\clist_put_left:No	101, 5166 , 5250	12, 881 , 885, 938, 979,
\clist_put_left:NV	101, 5166 , 5249	1181, 1187, 1493, 1524, 1595, 1637,
\clist_put_left:Nx	101, 5166	1650, 1651, 1653, 1655, 1657, 1658,
\clist_put_right:cn	102, 5174	1660, 1718, 1892, 1893, 1902–1904,
\clist_put_right:co	102, 5174	1906, 1908, 1910, 1912, 1914, 1916,
\clist_put_right:cV	102, 5174	1918–1929, 2646, 3142, 4242, 6276
\clist_put_right:Nn	102, 5174 ,	\cs_generate_from_arg_count:cNnn ...
5174, 5177, 5222, 5241, 5924, 6028	 1085, 1093,
\clist_put_right:No	102, 5174	1101, 1109, 1478, 1523, 6627, 6632
\clist_put_right:NV	102, 5174	\cs_generate_from_arg_count:NNnn ...
\clist_put_right:Nx	102, 5174 1449 , 1449 , 1479 , 1492
\clist_remove_duplicates:N		\cs_generate_from_arg_count_error_msg:Nn
.....	106 , 5215 , 5225 1449 , 1472 , 1481
\clist_remove_duplicates_aux:n		\cs_generate_internal_variant:n
.....	5215 , 5217, 5220 27, 1741, 1786 , 1786
\clist_remove_duplicates_aux:NN		\cs_generate_internal_variant_aux:n
.....	5215 , 5215, 5226, 5229 1789, 1792 , 1792, 1797
\clist_remove_element:Nn 106 , 5231 , 5231		\cs_generate_variant:Nn
\clist_remove_element_aux:n 27, 1719 , 1719, 1836–
.....	5231 , 5239, 5244, 5247	1855, 2025–2028, 2032, 2036, 2040,
\clist_remove_element_aux:NNn		2044, 2162, 2163, 2480, 2481, 2503–
.....	5231 , 5232, 5235, 5237	2510, 3053, 3065, 3066, 3070–3072,
\clist_set_eq:cc	100, 5089 , 5092	3076–3078, 3105–3108, 3110, 3112,
\clist_set_eq:cN	100, 5089 , 5090	3137–3140, 3468, 3690, 3799, 3813,
\clist_set_eq:Nc	100, 5089 , 5091	3814, 3828, 3829, 3836, 3850, 3859,
\clist_set_eq:NN 100 , 5089 , 5089, 5226, 5232		3864, 3869, 3871, 3919, 3923, 3925,
\clist_show:c	102, 5144 , 5145	3929, 3933, 3937, 3941, 3943, 3945,
\clist_show:N	102, 5144 , 5144	3949, 3951, 3953, 3955, 3957, 3959,
\clist_tmp:w	5110, 5114	4063, 4083, 4085, 4099–4115, 4130,
\clist_top:cN	107, 5260 , 5261	4131, 4133, 4135, 4147, 4158, 4189–
\clist_top:NN	107, 5260 , 5260	4197, 4228–4235, 4259–4262, 4266–
\clist_use:c	102, 5119	4269, 4288–4295, 4302–4309, 4339,
\clist_use:N	102, 5119 , 5119, 5127	4351, 4370, 4375, 4389, 4398, 4407,
\closein	210	4408, 4467–4469, 4476–4481, 4508,
\closeout	205	4509, 4539, 4540, 4543, 4544, 4551,
\clubpenalties	518	4552, 4556, 4558, 4563–4566, 4585–
\clubpenalty	345	4588, 4609, 4611, 4614, 4629, 4632,
.code:n	137	4649, 4650, 4660, 4661, 4670, 4671,
.code:x	137	4673, 4678, 4684, 4698, 4712–4714,
\copy	402	4725–4728, 4733–4736, 4782, 4783,
\count	453	4788–4791, 4796–4799, 4824, 4854,
\countdef	152	4865, 4866, 4895, 4911, 4912, 4936,
\cr	177	4963, 4964, 4992–4997, 5013, 5014,
\crrcr	178	5039, 5040, 5067, 5080, 5082, 5084,
\cs:w	12 , 881 , 884, 938, 979, 1187,	5086, 5088, 5116–5118, 5127, 5132,
1493, 1524, 1595, 1637, 1650, 1651,		5160, 5169, 5173, 5177, 5181, 5183–
		5187, 5195, 5212, 5213, 5253, 5257,
		5259, 5302, 5313, 5317, 5346, 5347,

5359, 5360, 5378–5380, 5392, 5402, 5421, 5431–5438, 5442–5445, 5447, 5449, 5451, 5453, 5456, 5458, 5464, 5466, 5470–5472, 5482–5484, 5486, 5488, 5490, 5499, 5501, 5503, 5505, 5509, 5511, 5524, 5526, 5529, 5531, 5535, 5537, 5541, 5546, 5553, 5555, 5615, 5616, 5620, 5624, 5637, 5650, 5756, 5768, 5800, 5804, 5815, 5819, 6646, 6656, 6765, 6772, 6834, 7410, 7422, 7429, 7430, 7466, 7467, 7514, 7515, 7531, 7596, 7600, 7674, 7889, 7893, 7925, 7929, 8010, 8011, 8038, 8039, 8067, 8068, 8171, 8172, 8192, 8193, 8304, 8305, 8828, 8829, 8927, 8928, 9127, 9128, 9401, 9402, 9694, 9737, 9738, 10065, 10066, 10518, 10522	\cs_gnew_protected:Npx 1349
\cs_generate_variant_aux:N 1719, 1731, 1736, 1746, 1753, 1754	\cs_gnew_protected_nopar:cpn 1352
\cs_generate_variant_aux:nnNn 1719, 1721, 1723	\cs_gnew_protected_nopar:cpx 1356
\cs_generate_variant_aux:nnw 1719, 1724, 1726, 1751	\cs_gnew_protected_nopar:Npn 1344
\cs_get_arg_count_from_signature:c 1446, 1525	\cs_gnew_protected_nopar:Npx 1348
\cs_get_arg_count_from_signature:N 24, 1012, 1018, 1025, 1032, 1430, 1430, 1447, 1494, 6679	\cs_gset:cn 21, 1528
\cs_get_arg_count_from_signature_aux:nnN 1430, 1431, 1433	\cs_gset:cpn 19, 1324, 1326, 2374, 2397, 4354, 4363, 5396
\cs_get_arg_count_from_signature_auxii:w 1430, 1440, 1445	\cs_gset:cpx . . . 19, 1324, 1327, 2353, 2361
\cs_get_function_name:N . 24, 1163, 1163	\cs_gset:cx 21, 1528
\cs_get_function_signature:N 24, 1163, 1166	\cs_gset:Nn 21, 1490
\cs_gnew:cpn 1351	\cs_gset:Npn 19, 918, 920, 1309, 1326, 1642, 4829, 10526
\cs_gnew:cpx 1355	\cs_gset:Npx 19, 918, 923, 1310, 1327, 4834
\cs_gnew:Npn 1343	\cs_gset:Nx 21, 1490
\cs_gnew:Npx 1347	\cs_gset_eq:cc 23, 1381, 1384, 2019
\cs_gnew_eq:cc 1388	\cs_gset_eq:cN 23, 1381, 1383, 1399, 2009, 2011, 2018, 2415, 4838, 5676, 5716
\cs_gnew_eq:cN 1386	\cs_gset_eq:Nc 23, 1381, 1382, 2017, 4845, 5667, 5685, 5707, 5725
\cs_gnew_eq:Nc 1387	\cs_gset_eq:NN . . 23, 1381, 1381–1384, 1396, 2008, 2010, 2016, 4122, 4128
\cs_gnew_eq:NN 1385	\cs_gset_nopar:cn 21, 1528
\cs_gnew_nopar:cpn 1350	\cs_gset_nopar:cpn . . 20, 1315, 1320, 1956
\cs_gnew_nopar:cpx 1354	\cs_gset_nopar:cpx . . 20, 1315, 1321, 2343
\cs_gnew_nopar:Npn 1342	\cs_gset_nopar:cx 21, 1528
\cs_gnew_nopar:Npx 1346	\cs_gset_nopar:Nn 21, 1490
\cs_gnew_protected:cpn 1353	\cs_gset_nopar:Npn 20, 918, 918, 921, 927, 933, 1307, 1320, 2627
\cs_gnew_protected:cpx 1357	\cs_gset_nopar:Npx 20, 918, 919, 924, 930, 936, 1308, 1321, 4094, 4097, 4175, 4178, 4181, 4184, 4187, 4214, 4217, 4220, 4223, 4226
\cs_gnew_protected:Npn 1345	\cs_gset_nopar:Nx 21, 1490
	\cs_gset_protected:cn 21, 1528
	\cs_gset_protected:cpn . . 20, 1336, 1338
	\cs_gset_protected:cpx 20, 1336, 1339, 2392
	\cs_gset_protected:cx 21, 1528
	\cs_gset_protected:Nn 21, 1490
	\cs_gset_protected:Npn 20, 918, 932, 1313, 1338
	\cs_gset_protected:Npx 20, 918, 935, 1314, 1339
	\cs_gset_protected:Nx 21, 1490
	\cs_gset_protected_nopar:cn . . 22, 1528
	\cs_gset_protected_nopar:cpn 20, 1330, 1332
	\cs_gset_protected_nopar:cpx 20, 1330, 1333, 2369

- \cs_gset_protected_nopar:cx ... [22](#), [1528](#)
- \cs_gset_protected_nopar:Nn ... [22](#), [1490](#)
- \cs_gset_protected_nopar:Npn [20](#), [918](#), [926](#), [1311](#), [1332](#)
- \cs_gset_protected_nopar:Npx [20](#), [918](#), [929](#), [1312](#), [1333](#)
- \cs_gset_protected_nopar:Nx ... [22](#), [1490](#)
- \cs_gundefine:c [22](#), [1389](#), [1398](#)
- \cs_gundefine:N [22](#), [1389](#), [1395](#), [5752](#), [5764](#)
- \cs_if_do_not_use_aux:nnN [1195](#), [1196](#), [1198](#)
- \cs_if_do_not_use_p:N [10](#), [1195](#), [1195](#), [1208](#)
- \cs_if_eq:ccF [1567](#)
- \cs_if_eq:ccT [1566](#)
- \cs_if_eq:ccTF [10](#), [1552](#), [1565](#)
- \cs_if_eq:cNF [1559](#)
- \cs_if_eq:cNT [1558](#)
- \cs_if_eq:cNTF [10](#), [1552](#), [1557](#)
- \cs_if_eq:NcF [1563](#)
- \cs_if_eq:NcT [1562](#)
- \cs_if_eq:NcTF [10](#), [1552](#), [1561](#)
- \cs_if_eq:NN [1552](#)
- \cs_if_eq:NNF [1559](#), [1563](#), [1567](#)
- \cs_if_eq:NNT [1558](#), [1562](#), [1566](#)
- \cs_if_eq:NNTF . [10](#), [1552](#), [1557](#), [1561](#), [1565](#)
- \cs_if_eq_name:NN [1297](#)
- \cs_if_eq_name_p:NN [10](#), [1297](#)
- \cs_if_eq_p:cc [10](#), [1552](#), [1564](#)
- \cs_if_eq_p:cN [10](#), [1552](#), [1556](#)
- \cs_if_eq_p:Nc [10](#), [1552](#), [1560](#)
- \cs_if_eq_p:NN . [10](#), [1552](#), [1556](#), [1560](#), [1564](#)
- \cs_if_exist:c [1180](#)
- \cs_if_exist:cF [3314](#), [3321](#), [3323](#)
- \cs_if_exist:cT [6815](#)
- \cs_if_exist:cTF [10](#), [1169](#),
[5666](#), [5695](#), [5706](#), [5735](#), [6029](#), [6036](#),
[6092](#), [6191](#), [6266](#), [6604](#), [6711](#), [10298](#)
- \cs_if_exist:N [1169](#)
- \cs_if_exist:Nf [1278](#), [6579](#), [6827](#)
- \cs_if_exist:NT .. [5746](#), [5758](#), [7049](#), [7066](#)
- \cs_if_exist:NTF [10](#), [1169](#), [1402](#),
[1411](#), [1418](#), [2696](#), [5824](#), [6669](#), [6704](#)
- \cs_if_exist_p:c [10](#), [1169](#)
- \cs_if_exist_p:N [10](#), [1169](#), [1202](#), [4139](#), [4151](#)
- \cs_if_free:cF [1219](#)
- \cs_if_free:cT [1218](#), [1787](#), [2367](#), [6637](#), [6642](#)
- \cs_if_free:cTF
.... [10](#), [1201](#), [1217](#), [1730](#), [2335](#), [6285](#)
- \cs_if_free:N [1201](#)
- \cs_if_free:Nf [1219](#), [1247](#), [1266](#)
- \cs_if_free:NT [1218](#)
- \cs_if_free:NTF [10](#),
[1201](#), [1217](#), [5798](#), [5802](#), [7412](#), [10535](#)
- \cs_if_free_p:c [10](#), [1201](#), [1216](#)
- \cs_if_free_p:N [10](#), [1201](#), [1216](#)
- \cs_meaning:c [12](#), [881](#), [887](#)
- \cs_meaning:N [12](#), [881](#), [886](#), [887](#)
- \cs_new:cn [18](#), [1544](#)
- \cs_new:cpn [17](#),
[1324](#), [1328](#), [1351](#), [1902–1905](#), [1907](#),
[1909](#), [1911](#), [1913](#), [1915](#), [1917](#), [1919–](#)
[1929](#), [2069](#), [2081](#), [2115](#), [2118](#), [2121](#),
[2122](#), [2143–2146](#), [3992](#), [3996](#), [4000](#),
[4004](#), [4008](#), [4012](#), [4016](#), [6754](#), [6757](#)
- \cs_new:cpx [17](#), [1324](#), [1329](#), [1355](#), [1788](#)
- \cs_new:cx [18](#), [1544](#)
- \cs_new:Nn [18](#), [1513](#)
- \cs_new:Npn [17](#),
[993](#), [1017](#), [1300](#), [1309](#), [1328](#), [1343](#),
[1581](#), [1584](#), [1587](#), [1588](#), [1591](#), [1594](#),
[1597](#), [1600](#), [1610](#), [1618](#), [1645](#), [1646](#),
[1648](#), [1651](#), [1652](#), [1654](#), [1656](#), [1659](#),
[1708](#), [1709](#), [1712](#), [1715](#), [1718](#), [1775](#),
[1792](#), [1799](#), [1800](#), [1804](#), [1807](#), [1811](#),
[1827](#), [1833](#), [1930](#), [1936](#), [1945](#), [1964](#),
[1972](#), [2029](#), [2033](#), [2037](#), [2041](#), [2045](#),
[2049](#), [2056](#), [2063](#), [2066](#), [2093](#), [2096](#),
[2099](#), [2107](#), [2147](#), [2150](#), [2155](#), [2158](#),
[2159](#), [2164](#), [2165](#), [2173](#), [2176](#), [2179](#),
[2182](#), [2185](#), [2189](#), [2195](#), [2199](#), [2205](#),
[2209](#), [2215](#), [2219](#), [2225](#), [2433](#), [2439](#),
[2448](#), [2461](#), [2477](#), [2900](#), [2903](#), [2933](#),
[3345](#), [3357](#), [3374](#), [3406](#), [3414](#), [3417](#),
[3426](#), [3434](#), [3438](#), [3442](#), [3446](#), [3469](#),
[3472](#), [3475](#), [3478](#), [3481](#), [3484](#), [3487](#),
[3495](#), [3498](#), [3506](#), [3983](#), [3987](#), [4299](#),
[4330](#), [4340](#), [4347](#), [4382](#), [4385](#), [4393](#),
[4404](#), [4553](#), [4555](#), [4557](#), [4559](#), [4561](#),
[4562](#), [4576](#), [4685](#), [4753](#), [4800](#), [4801](#),
[4812](#), [4818](#), [4931](#), [5061](#), [5066](#), [5143](#),
[5247](#), [5459–5462](#), [5821](#), [5845](#), [5846](#),
[6043](#), [6406](#), [6409](#), [6412](#), [6435](#), [6465](#),
[6486](#), [6527](#), [6528](#), [6811](#), [10512](#), [10515](#)
- \cs_new:Npx [17](#), [1300](#), [1310](#), [1329](#), [1347](#)
- \cs_new:Nx [18](#), [1513](#)
- \cs_new_eq:cc .. [22](#), [1038](#), [1374](#), [1380](#), [1388](#)
- \cs_new_eq:cN [22](#), [1374](#),
[1378](#), [1386](#), [2003](#), [9179](#), [9202](#), [9232](#)
- \cs_new_eq:Nc [22](#), [1374](#), [1379](#), [1387](#)
- \cs_new_eq:NN [22](#),
[1374](#), [1374](#), [1378–1380](#), [1385](#), [1412](#),

- 1413, 1419, 1420, 2002, 2012–2019,
2517, 2563, 2579, 2589, 2599, 2611–
2613, 3036, 3038–3040, 3141, 3143,
3144, 3858, 3863, 3868, 3870, 3956,
3958, 3966, 3967, 3971, 4047, 4048,
4127, 4128, 4146, 4157, 4333–4335,
4381, 4554, 4560, 4610, 4617, 4631,
4646, 4647, 4672, 4760–4777, 4802–
4804, 4867–4886, 4965–4970, 5068–
5071, 5079, 5081, 5083, 5085, 5087,
5089–5096, 5144, 5145, 5248–5251,
5254–5256, 5260, 5261, 5270–5285,
5403, 5422–5424, 5454, 5467–5469,
5485, 5487, 5492, 5514, 5517, 5523,
5525, 5542, 5547, 5552, 5554, 5563–
5566, 5586, 5598, 5618, 5622, 5822,
7518–7527, 10508, 10509, 10609–10612
\cs_new_nopar:cn 18, [1544](#)
\cs_new_nopar:cpn
..... 17, [1315](#), 1322, 1350, 2133–
2142, 7567, 7568, 7571, 7574, 7577,
7580, 7583, 7586, 7589, 7633, 7636,
7639, 7642, 7645, 7648, 7651, 7655,
7659, 7702, 7706, 7710, 7714, 7718,
7722, 7726, 7730, 7734, 7738, 7743
\cs_new_nopar:cpx
..... 17, [1315](#), 1323, 1354, 1782, 2947
\cs_new_nopar:cx 18, [1544](#)
\cs_new_nopar:Nn 18, [1513](#)
\cs_new_nopar:Npn 17, [1300](#),
1307, 1322, 1342, 1401, 1410, 1429,
1556–1567, 1580, 1605, 1768, 1773,
1817–1821, 1824, 1830, 1876, 1878,
1879, 1889, 1895, 1898, 1901, 2123,
2127, 2131, 2132, 2309, 2313, 2322,
2326, 2521, 2522, 2525, 2528, 2571,
2572, 2575, 2576, 2583, 2584, 2586,
2587, 2593, 2594, 2596, 2597, 2603,
2604, 2607, 2608, 2688, 2718, 2725,
2736, 2747, 2758, 2769, 2776, 2783,
2790, 2804, 2807, 2813, 2819, 2845,
2875, 2910, 2917, 2924, 2941, 3015,
3119, 3142, 3145, 3158, 3162, 3166,
3170, 3191, 3222, 3253, 3264, 3277,
3300, 3308, 3341, 3344, 3851, 3860,
3865, 3872, 3898, 3968, 4020, 4023,
4026, 4029, 4075, 4084, 4336, 4338,
4343, 4399, 4991, 5119, 5375, 5820,
5894, 5897, 5902, 5903, 6042, 6249–
6251, 6265, 6284, 6400, 6717, 6753,
6808, 7528, 7532, 7549, 7550, 7555,
7563, 7590, 7595, 7597, 7601, 7612,
7622, 7630, 7660, 7667, 7671, 7675,
7686, 7694, 7697, 7744, 7758, 7777,
7797, 7802, 7840, 7878–7880, 7883
\cs_new_nopar:Npx 17, [1300](#), 1308, 1323, 1346
\cs_new_nopar:Nx 18, [1513](#)
\cs_new_protected:cn 18, [1544](#)
\cs_new_protected:cpn 17, [1336](#), 1340, 1353
\cs_new_protected:cpx 17, [1336](#),
1341, 1357, 6149, 6154, 6159, 6163
\cs_new_protected:cx 18, [1544](#)
\cs_new_protected:Nn 18, [1513](#)
\cs_new_protected:Npn 17, 1005, 1031,
[1300](#), 1313, 1340, 1345, 1374, 1381,
1606, 1719, 1786, 1954, 1980, 1986,
1994, 2334, 2342, 2877, 2885, 2888,
2891, 4056, 4064, 4071, 4087, 4090,
4093, 4096, 4159, 4162, 4165, 4168,
4171, 4174, 4177, 4180, 4183, 4186,
4198, 4201, 4204, 4207, 4210, 4213,
4216, 4219, 4222, 4225, 4352, 4361,
4371, 4376, 4405, 4406, 4418, 4434,
4444, 4452, 4486, 4516, 4541, 4542,
4545, 4548, 4618, 4621, 4624, 4625,
4689, 4700, 4706, 4716, 4784, 4786,
4792, 4794, 4826, 4831, 4836, 4848,
4855, 4893, 4906, 4999, 5001, 5003,
5015, 5017, 5019, 5133, 5134, 5138,
5161, 5188, 5196, 5215, 5220, 5237,
5303, 5309, 5312, 5314, 5316, 5318,
5320, 5326, 5332, 5338, 5348, 5350,
5352, 5362, 5364, 5416, 5498, 5502,
5506, 5518, 5519, 5528, 5532, 5548–
5551, 5935, 5958, 5970, 5977, 6019,
6064, 6168, 6181, 6374, 6413, 6421,
6436, 6447, 6466, 6482, 6487, 6513,
6584, 6601, 6625, 6630, 6641, 6647,
6650, 6661, 6665, 6676, 6762, 6766,
6777, 6781, 6812, 10513, 10519, 10594
\cs_new_protected:Npx
..... 17, [1300](#), 1314, 1341, 1349
\cs_new_protected:Nx 18, [1513](#)
\cs_new_protected_nopar:cn 19, [1544](#)
\cs_new_protected_nopar:cpn
..... 18, [1330](#), 1334, 1352, 10305,
10330, 10364, 10381, 10412, 10443
\cs_new_protected_nopar:cpx
..... 18, [1330](#), 1335, 1356, 1780
\cs_new_protected_nopar:cx 19, [1544](#)

- `\cs_new_protected_nopar:Nn` [19](#), [1513](#)
`\cs_new_protected_nopar:Npn` [18](#), [1300](#),
1311, 1316, 1334, 1344, 1378–1380,
1382–1384, 1389, 1392, 1395, 1398,
2002–2011, 2226, 2425, 2518, 2531–
2562, 2564, 2567, 2568, 2580, 2590,
2600, 2611, 2869, 2870, 2946, 3018,
3048, 3054, 3059, 3067, 3073, 3079,
3084, 3089, 3095, 3109, 3111, 3113,
3125, 3131, 3146, 3671, 3794, 3800,
3806, 3815, 3821, 3830, 3837, 3843,
3914, 3920, 3924, 3926, 3930, 3934,
3938, 3942, 3944, 3946, 3950, 3952,
3954, 4036, 4041–4046, 4069, 4070,
4117, 4121, 4132, 4134, 4137, 4149,
4236, 4242, 4374, 4390, 4483, 4505,
4510, 4513, 4604, 4613, 4630, 4634,
4639, 4651, 4655, 4662, 4665, 4674,
4679, 4693, 4778, 4780, 4805, 4842,
4887, 4896, 4898, 4900, 4913, 4919,
4937, 4939, 4941, 4947, 5042, 5099,
5128, 5146, 5166, 5170, 5174, 5178,
5194, 5201, 5210, 5211, 5225, 5228,
5231, 5234, 5252, 5258, 5286, 5394,
5446, 5448, 5450, 5452, 5455, 5457,
5463, 5465, 5473, 5476, 5479, 5496,
5497, 5500, 5504, 5510, 5512, 5515,
5520, 5527, 5530, 5536, 5538, 5543,
5602, 5606, 5617, 5621, 5625, 5638,
5651, 5658, 5665, 5690, 5705, 5730,
5745, 5757, 5769, 5772, 5789, 5790,
5794, 5796, 5797, 5801, 5805, 5808,
5811, 5816, 5828, 5831, 5904, 5908,
5912, 5917, 5922, 5984, 5987, 5990,
5993, 5996, 6001, 6044, 6053, 6127,
6130, 6133, 6136, 6139, 6142, 6145,
6148, 6201, 6205, 6279, 6365, 6368,
6371, 6397, 6529, 6533, 6537, 6568,
6587, 6592, 6616, 6635, 6657, 6691,
6696, 6702, 6722, 6726, 6731, 6739,
6742, 6756, 6759, 6773, 6796, 6822,
6826, 7035, 7045, 7078, 7095, 7100,
7102, 7200, 7203, 7213, 7222, 7247,
7255, 7258, 7261, 7266, 7269, 7285,
7296, 7350–7352, 7359, 7368, 7382,
7391, 7405, 7406, 7411, 7423, 7426,
7431, 7434, 7437, 7468, 7471, 7474,
7886, 7890, 7894, 7922, 7926, 7930,
7959, 7967, 7979, 8004, 8007, 8012,
8032, 8035, 8040, 8061, 8064, 8069,
8081, 8118, 8134, 8165, 8168, 8173,
8186, 8189, 8194, 8244, 8273, 8284,
8288, 8293, 8298, 8301, 8306, 8341,
8387, 8405, 8421, 8431, 8432, 8436,
8444, 8452, 8467, 8489, 8532, 8591,
8607, 8611, 8620, 8629, 8636, 8645,
8667, 8676, 8689, 8694, 8706, 8724,
8741, 8779, 8796, 8822, 8825, 8830,
8876, 8909, 8921, 8924, 8929, 8964,
8997, 9019, 9054, 9068, 9121, 9124,
9129, 9175, 9184, 9212, 9242, 9395,
9398, 9403, 9446, 9473, 9484, 9518,
9549, 9568, 9573, 9579, 9643, 9690,
9731, 9734, 9739, 9773, 9796, 9830,
9861, 9874, 9883, 9905, 9930, 9937,
9948, 9953, 10004, 10013, 10028,
10059, 10062, 10067, 10117, 10152,
10206, 10217, 10297, 10472, 10481,
10487, 10496, 10534, 10565, 10574
`\cs_new_protected_nopar:Npx`
. [18](#), [1300](#), 1312, 1335, 1348
`\cs_new_protected_nopar:Nx` [19](#), [1513](#)
`\cs_record_meaning:N` [16](#), [1243](#), 1244, 1254
`\cs_set:cn` [21](#), [1528](#)
`\cs_set:cpn` [19](#),
[1324](#), 1324, 3524, 3528, 3532, 3536,
3540, 3544, 3548, 5914, 5915, 5919
`\cs_set:cpx` [19](#), [1324](#),
1325, 2227, 2231, 2235, 2239, 2243,
2252, 2261, 2270, 2279, 2281, 2283,
2285, 2287, 2289, 2291, 2293, 2295
`\cs_set:cx` [21](#), [1528](#)
`\cs_set:Nn` [21](#), [1490](#)
`\cs_set:Npn` [19](#), [898](#), 900, 938, 946–
979, 987, 1011, 1040–1042, 1047,
1112, 1115, 1126–1129, 1132, 1146,
1150, 1159, 1163, 1166, 1195, 1198,
1244, 1300, 1315, 1324, 1430, 1433,
1445, 1449, 1481, 1490, 1521, 1650,
1723, 1726, 1753, 2302, 2303, 3041,
3514, 3518, 3578, 3585, 3595, 3605,
3608, 3632, 3633, 3647, 3650, 3653,
3656, 3659, 3662, 3665, 3668, 4428,
4462, 4471, 5110, 5239, 5304, 5385,
5946, 5972, 5979, 6024, 6454, 6628
`\cs_set:Npx` . . [19](#), [898](#), 903, 941, 1325, 6633
`\cs_set:Nx` [21](#), [1490](#)
`\cs_set_eq:cc` . . [23](#), 1035, [1370](#), 1373, 2015
`\cs_set_eq:cN` [23](#), [1370](#), 1371, 1393, 2005,
2007, 2014, 5920, 6375, 6378, 6760

- \cs_set_eq:Nc [23](#), [1370](#),
1372, 2013, 6386, 6388, 6667, 6703
- \cs_set_eq:NN . . . [23](#), [1370](#), 1370–1373,
1376, 1381, 1385–1388, 1390, 2004,
2006, 2012, 2878, 2892, 2908, 2949,
3021, 3032, 3033, 3035, 3037, 3738–
3744, 3746–3749, 3751–3754, 3756–
3758, 3760–3762, 3764–3767, 3769–
3772, 3774–3777, 3779–3782, 3889,
3890, 3972, 3973, 4086, 4118, 4127,
5489, 5491, 5611, 5612, 5814, 5938,
5973, 5980, 6323, 6381, 6382, 6530,
6534, 6538, 8344, 8434, 9245, 10250
- \cs_set_eq:NwN [23](#),
[853](#), 859–886, 888, 891–899, 918,
919, 1221, 1342–1357, 1370, 1576–1578
- \cs_set_nopar:cn [21](#), [1528](#)
- \cs_set_nopar:cpn . . . [19](#), [1315](#), 1318,
1358–1366, 1368, 3147, 3148, 6258
- \cs_set_nopar:cpx [19](#), [1315](#), 1319
- \cs_set_nopar:cx [21](#), [1528](#)
- \cs_set_nopar:Nn [21](#), [1490](#)
- \cs_set_nopar:Npn . . [19](#), 842, [898](#), 898,
900, 901, 903, 906, 907, 909, 913,
980, 982, 1139, 1216–1219, 1318,
1446, 1478, 1626, 1636, 1661–1663,
1665–1667, 1671–1676, 1679, 1680,
1683, 1687, 1689, 1691, 1694, 1695,
1699–1702, 1707, 1989, 1997, 3181,
3295, 3297, 3557, 3564, 3571, 5197,
5323, 5342, 5353, 5355, 5381, 5937,
5943, 5959, 6020, 6274, 6290, 6304,
6305, 6314, 6315, 6318, 6321, 7510
- \cs_set_nopar:Npx [19](#), [898](#), 899,
904, 910, 916, 1319, 1607, 4088,
4091, 4160, 4163, 4166, 4169, 4172,
4199, 4202, 4205, 4208, 4211, 6291
- \cs_set_nopar:Nx [21](#), [1490](#)
- \cs_set_protected:cn [21](#), [1528](#)
- \cs_set_protected:cpn [20](#), [1336](#), 1336, 6003
- \cs_set_protected:cpx [20](#), [1336](#), 1337,
1491, 1522, 6006, 6009, 6012, 6015
- \cs_set_protected:cx [21](#), [1528](#)
- \cs_set_protected:Nn [21](#), [1490](#)
- \cs_set_protected:Npn
. [20](#), [898](#), 912, 939, 984,
990, 996, 999, 1002, 1008, 1014,
1021, 1024, 1028, 1034, 1037, 1054,
1062, 1070, 1078, 1083, 1091, 1099,
1107, 1336, 1370, 3034, 3974, 4487,
4492, 4518, 4522, 4980, 6232, 6278
- \cs_set_protected:Npx . . [20](#), [898](#), 915, 1337
- \cs_set_protected:Nx [21](#), [1490](#)
- \cs_set_protected_nopar:cn [22](#), [1528](#)
- \cs_set_protected_nopar:cpn
. [20](#), [1330](#), 1330, 6255
- \cs_set_protected_nopar:cpx
. [20](#), [1330](#), 1331
- \cs_set_protected_nopar:cx [22](#), [1528](#)
- \cs_set_protected_nopar:Nn [22](#), [1490](#)
- \cs_set_protected_nopar:Npn
. [20](#), [898](#), 906, 912, 915, 920,
923, 926, 929, 932, 935, 1233, 1236,
1239, 1246, 1260, 1265, 1277, 1284,
1301, 1330, 1371–1373, 1664, 1668,
1670, 1677, 1681, 1685, 1686, 1690,
1692, 1696, 1697, 1703, 1704, 1706,
3101–3104, 5793, 5795, 7297, 7306,
7314, 7323, 8275, 10554, 10558,
10582, 10583, 10587, 10589, 10601
- \cs_set_protected_nopar:Npx
. [20](#), [898](#), 909, 1331,
7446, 7489, 7509, 7902, 7939, 8016,
8098, 8213, 8319, 8329, 8356, 8854,
8867, 8956, 9153, 9166, 9431, 9755,
9764, 9789, 9975, 10088, 10094,
10105, 10137, 10144, 10189, 10223
- \cs_set_protected_nopar:Nx [22](#), [1490](#)
- \cs_show:c [12](#), [881](#), 889, 6809
- \cs_show:N [12](#), [881](#), 888, 889, 4084
- \cs_split_function:NN
. [24](#), 986, 992, 998, 1004, 1010,
1016, 1023, 1030, 1120, 1122, [1141](#),
1146, 1164, 1167, 1196, 1431, 1721
- \cs_split_function_aux:w [1141](#), 1147, 1150
- \cs_split_function_auxii:w
. [1141](#), 1157, 1159
- \cs_tmp: 941, 944
- \cs_tmp:w 1300, 1307–1315, 1318–
1341, 1490, 1497–1521, 1528–1551
- \cs_to_str:N
. [4](#), [24](#), [1132](#), 1132, 1148, 2311, 5821
- \cs_to_str_aux:w [1132](#), 1135, 1139
- \cs_undefine:c [22](#), [1389](#), 1392
- \cs_undefine:N [22](#), [1389](#), 1389
- \csname 12, 31, 34, 52, 58, 62, 240
- \currentgrouplevel 492
- \currentgrouptype 493
- \currentifbranch 489

<code>\currentiflevel</code>	488	<code>\dim_gset:cn</code>	72, 3920
<code>\currentiftyp</code>	490	<code>.dim_gset:N</code>	138
D			
<code>\d</code>	1757, 2708	<code>\dim_gset:Nc</code>	72, 3920
<code>\dagger</code>	3267, 3273	<code>\dim_gset:Nn</code> 72, 3920, 3924, 3925, 3935, 3939	
<code>\day</code>	448	<code>\dim_gset_max:cn</code>	72, 3926
<code>\ddagger</code>	3268, 3274	<code>\dim_gset_max:Nn</code> ...	72, 3926, 3934, 3937
<code>\deadcycles</code>	382	<code>\dim_gset_min:cn</code>	72, 3926
<code>\DeclareOption</code>	130, 133	<code>\dim_gset_min:Nn</code> ...	72, 3926, 3938, 3941
<code>\def</code>	45, 46, 98, 125, 131, 134, 139, 147	<code>\dim_gsub:cn</code>	73, 3952
<code>\def_long_test_function_new:npn</code> ..	4583	<code>\dim_gsub:Nn</code>	73, 3952, 3954, 3955
<code>.default:n</code>	137	<code>\dim_gzero:c</code>	71, 3942
<code>.default:V</code>	137	<code>\dim_gzero:N</code>	71, 3942, 3944, 3945
<code>\default@ds</code>	812	<code>\dim_new:c</code>	71, 3910
<code>\defaultthyphenchar</code>	432	<code>\dim_new:N</code>	71, 3910, 3914, 3919, 3960–3965, 7516
<code>\defaultskewchar</code>	433	<code>.dim_set:c</code>	138
<code>\DefineCrossReferences</code>	6305, 6311	<code>\dim_set:cn</code>	72, 3920
<code>\delcode</code>	463	<code>.dim_set:N</code>	138
<code>\delimiter</code>	257	<code>\dim_set:Nc</code>	72, 3920
<code>\delimiterfactor</code>	306	<code>\dim_set:Nn</code>	72, 3920, 3920, 3923, 3924, 3927, 3931
<code>\delimitershortfall</code>	305	<code>\dim_set_max:cn</code>	72, 3926
<code>\detokenize</code>	480	<code>\dim_set_max:Nn</code> ...	72, 3926, 3926, 3929
<code>\dim_add:cn</code>	72, 3946	<code>\dim_set_min:cn</code>	72, 3926
<code>\dim_add:Nc</code>	72, 3946	<code>\dim_set_min:Nn</code> ...	72, 3926, 3930, 3933
<code>\dim_add:Nn</code>	72, 3946, 3946, 3949, 3950	<code>\dim_show:c</code>	73, 3958
<code>\dim_compare:n</code>	3979	<code>\dim_show:N</code>	73, 3958, 3958, 3959
<code>\dim_compare:nNn</code>	3975	<code>\dim_sub:cn</code>	73, 3952
<code>\dim_compare:nNnF</code>	4024, 4030	<code>\dim_sub:Nc</code>	73, 3952
<code>\dim_compare:nNnT</code>		<code>\dim_sub:Nn</code>	73, 3952, 3952–3954
..	3927, 3931, 3935, 3939, 4021, 4027	<code>\dim_until_do:nNnn</code> ..	74, 4020, 4023, 4024
<code>\dim_compare:nNnTF</code>	74, 2201, 3975	<code>\dim_use:c</code>	73, 3956
<code>\dim_compare:nTF</code>	74, 3979	<code>\dim_use:N</code>	73, 3956, 3956, 3957, 7481
<code>\dim_compare_auxi:w</code>	3980, 3983	<code>\dim_value:w</code>	3971, 3972, 3980
<code>\dim_compare_auxii:w</code>	3984, 3987	<code>\dim_while_do:nNnn</code> ..	74, 4020, 4020, 4021
<code>\dim_compare_p:n</code>	74, 3979	<code>\dim_zero:c</code>	71, 3942
<code>\dim_compare_p:nNn</code>	3975	<code>\dim_zero:N</code>	71, 3942, 3942–3944
<code>\dim_compare_p:nNnTF</code>	74	<code>\dimen</code>	454
<code>\dim_do_until:nNnn</code> ..	74, 4020, 4029, 4030	<code>\dimendef</code>	153
<code>\dim_do_while:nNnn</code> ..	74, 4020, 4026, 4027	<code>\dimexpr</code>	507
<code>\dim_eval:n</code>	73, 2196, 3968, 3968, 5459–5462, 5518, 5533	<code>\directlua</code>	14, 637
<code>\dim_eval:w</code> ...	3971, 3973, 3981, 3993, 3997, 4001, 4005, 4009, 4013, 4017	<code>\discretionary</code>	317
<code>\dim_eval_end:</code>	3971, 3974, 3993, 3997, 4001, 4005, 4009, 4013, 4017	<code>\displayindent</code>	282
<code>\dim_gadd:cn</code>	72, 3946	<code>\displaylimits</code>	292
<code>\dim_gadd:Nn</code>	72, 3946, 3950, 3951	<code>\displaystyle</code>	270
<code>.dim_gset:c</code>	138	<code>\displaywidowpenalties</code>	520
<code>\dim_gset:cc</code>	72, 3920	<code>\displaywidowpenalty</code>	281
		<code>\displaywidth</code>	283
		<code>\divide</code>	160
		<code>\documentclass</code>	6301

<code>\doublehyphendemerits</code>	350	<code>\etex_clubpenalties:D</code>	518
<code>\dp</code>	461	<code>\etex_currentgrouplevel:D</code>	492
<code>\ds@</code>	813	<code>\etex_currentgroupstype:D</code>	493, 1880
<code>\dump</code>	444	<code>\etex_currentifbranch:D</code>	489
E			
<code>\E</code>	1763, 2710	<code>\etex_currentiflevel:D</code>	488
<code>\edef</code>	32, 96, 148	<code>\etex_currentifttype:D</code>	490
<code>\efcode</code>	545	<code>\etex_detokenize:D</code>	480, 1779, 2452, 2465, 4335
<code>\else</code>	13, 53, 201, 1264	<code>\etex_dimexpr:D</code>	507, 3921, 3947, 3969, 3973, 3976, 5474, 5477, 5480
<code>\else:</code>	8, 846, 860, 863, 967, 1153, 1172, 1175, 1183, 1189, 1204, 1210, 1291, 1295, 1416, 1423, 1427, 1436, 1470, 1554, 1631, 1864, 1867, 1870, 1874, 2023, 2052, 2059, 2072, 2075, 2084, 2087, 2103, 2111, 2171, 2484, 2490, 2494, 2500, 2631, 2635, 2639, 2643, 2647, 2651, 2655, 2659, 2663, 2667, 2671, 2675, 2679, 2683, 2690, 2694, 2698, 2731, 2742, 2753, 2764, 2830, 2833, 2837, 2840, 2847, 2849, 2851, 2853, 2855, 2857, 2913, 2920, 2936, 3516, 3526, 3530, 3534, 3538, 3542, 3546, 3550, 3555, 3590, 3600, 3612, 3617, 3620, 3623, 3641, 3645, 3977, 3985, 3994, 3998, 4002, 4006, 4010, 4014, 4018, 4079, 4141, 4153, 4257, 4264, 4286, 4300, 4569, 4574, 4579, 4592, 5112, 5123, 5426, 5429, 5440, 5825	<code>\etex_displaywidowpenalties:D</code>	520
<code>\emergencystretch</code>	365	<code>\etex_endL:D</code>	528
<code>\end</code>	239, 6348	<code>\etex_endR:D</code>	530
<code>\EndCatcodeRegime</code>	10590	<code>\etex_eTeXrevision:D</code>	472
<code>\endcsname</code>	12, 31, 34, 52, 58, 62, 241	<code>\etex_eTeXversion:D</code>	471
<code>\endgroup</code>	11, 51, 57, 61, 174	<code>\etex_everyeof:D</code> 532, 4420, 4436, 4446, 4454	
<code>\endinput</code>	93, 213	<code>\etex_firstmarks:D</code>	475
<code>\endL</code>	528	<code>\etex_fontchardp:D</code>	500
<code>\endlinechar</code>	103, 115, 255	<code>\etex_fontcharht:D</code>	499
<code>\endR</code>	530	<code>\etex_fontcharic:D</code>	502
<code>\eqno</code>	275	<code>\etex_fontcharwd:D</code>	501
<code>\errhelp</code>	79, 221	<code>\etex_glueexpr:D</code>	508, 3801, 3831, 3838, 3861, 3866, 3873, 7476
<code>\errmessage</code>	215	<code>\etex_glueshrink:D</code>	511, 3907
<code>\erroneous</code>	1643	<code>\etex_glueshrinkorder:D</code>	513, 3895
<code>\ERROR</code> ..	1919, 2302, 2303, 6465, 6527, 6528	<code>\etex_gluestretch:D</code>	510, 3906
<code>\errorcontextlines</code>	222	<code>\etex_gluestretchorder:D</code>	512, 3894
<code>\errorstopmode</code>	236	<code>\etex_gluetomu:D</code>	514
<code>\escapechar</code>	254	<code>\etex_ifcsname:D</code>	469, 877, 8860, 8952, 9159, 9427, 9436, 9785
<code>\etex_beginL:D</code>	527	<code>\etex_ifdefined:D</code>	468, 876
<code>\etex_beginR:D</code>	529	<code>\etex_iffontchar:D</code>	498
<code>\etex_botmarks:D</code>	476	<code>\etex_interactionmode:D</code>	496
		<code>\etex_interlinepenalties:D</code>	517
		<code>\etex_lastlinefit:D</code>	516
		<code>\etex_lastnodetype:D</code>	497, 1882, 1884
		<code>\etex_marks:D</code>	473
		<code>\etex_middle:D</code>	521
		<code>\etex_muexpr:D</code>	509, 4041, 4043, 4045
		<code>\etex_mutoglua:D</code>	515
		<code>\etex_numexpr:D</code>	506, 1450, 1484, 3033, 3304, 3558, 3565, 3572, 3682, 7250, 7253, 7272, 7287, 7664, 7761, 7766, 7780, 7785, 7982, 7983, 8084, 8111, 8121, 8125, 8136, 8140, 8153, 8157, 8205, 8219, 8223, 8236, 8286, 8316, 8326, 8348, 8362, 8366, 8379, 8396, 8438, 8447, 8453– 8455, 8468–8470, 8480, 8484, 8485, 8594, 8601, 8623, 8632, 8743, 8745,

8747, 8759, 8765, 8769, 8773, 8848, 8902, 8947, 8990, 9147, 9249, 9268, 9422, 9498, 9530, 9589, 9595, 9599, 9636, 9655, 9749, 9780, 9823, 9844, 9924, 10035, 10081, 10085, 10102, 10127, 10198, 10331, 10334, 10349	
\etex_pagediscards:D	524
\etex_parshapedimen:D	505
\etex_parshapeindent:D	503
\etex_parshapelength:D	504
\etex_predisplaydirection:D	531
\etex_protected:D	533, 648, 651, 682, 686, 690, 695, 714, 723, 730, 738, 748, 752, 756, 776, 830, 897
\etex_readline:D	483
\etex_savinghyphcodes:D	522
\etex_savingvdiscards:D	523
\etex_scantokens:D	481, 4423, 4439, 4449, 4457
\etex_showgroups:D	494
\etex_showifs:D	495
\etex_showtokens:D	482, 4086, 5057
\etex_splitbotmarks:D	478
\etex_splitdiscards:D	525
\etex_splitfirstmarks:D	477
\etex_TeXxETstate:D	526
\etex_topmarks:D	474
\etex_tracingassigns:D	484
\etex_tracinggroups:D	491
\etex_tracingifs:D	487
\etex_tracingnesting:D	486
\etex_tracingscantokens:D	485
\etex_unexpanded:D	479, 652, 761, 765, 768, 880, 1289, 1578
\etex_unless:D	470, 865
\etex_widowpenalties:D	519
\eTeXrevision	472
\eTeXversion	471
\everycr	183
\everydisplay	284
\everyeof	532
\everyhbox	423
\everyjob	30, 452
\everymath	308
\everypar	371
\everyvbox	424
\ExecuteOptions	135
\exhyphenpenalty	347
\exp_after:cc	6274
\exp_after:wN	15, 209, 878, 878, 942, 979, 981, 983, 1049, 1117, 1137, 1147, 1152, 1154, 1182, 1184, 1187, 1203, 1205, 1435, 1437, 1440, 1493, 1524, 1575, 1576, 1595, 1598, 1601, 1602, 1608, 1611, 1612, 1619, 1620, 1627, 1632, 1634, 1637, 1645–1652, 1654–1657, 1659, 1660, 1708, 1710, 1713, 1716, 1718, 1728, 1754, 1770, 1778, 1780, 1782, 1795, 1801, 1802, 1805, 1808, 1809, 1812, 1813, 1828, 1834, 1891, 2071, 2074, 2076, 2083, 2086, 2088, 2094, 2097, 2100, 2108, 2116, 2119, 2121, 2122, 2124, 2128, 2323, 2347, 2349, 2436, 2442, 2444, 2452, 2458, 2465, 2471, 2473, 2646, 2686, 2697, 2715, 2722, 2732, 2743, 2754, 2765, 2773, 2780, 2787, 2809, 2815, 2821, 2912, 2914, 2919, 2921, 2931, 2935, 2937, 3159, 3163, 3511, 3515, 3980, 3984, 4080, 4285, 4297, 4327, 4336, 4344, 4345, 4423, 4428–4431, 4457, 4465, 4500, 4533, 4568, 4572, 4577, 4590, 4619, 4622, 4624, 4626, 4675, 4686, 4690, 4701, 4702, 4707, 4708, 4717, 4718, 4809, 4814, 4890, 4903, 4922, 4923, 4951, 4952, 4984, 5057, 5058, 5124, 5130, 5136, 5307, 5382, 5383, 5387, 6268, 6275, 6276, 6418, 6430, 6444, 6451, 6472, 6473, 6490, 6491, 6501, 6506, 6724, 7012, 7024
\exp_arg_last_unbraced:nn	1799, 1799, 1801, 1805, 1808, 1812
\exp_arg_next:nnn	1581, 1581, 1598, 1601, 1608, 1611, 1619
\exp_arg_next_nobraces:nnn	1581, 1584, 1595
\exp_args:cc	29, 1650, 1651
\exp_args:Nc	29, 887, 889, 890, 979, 979, 1056, 1064, 1072, 1080, 1216– 1219, 1261, 1285, 1316, 1371, 1378, 1383, 1410, 1447, 1479, 1556–1559, 1650, 1650, 1774, 2404, 4356, 6679
\exp_args:Ncc	30, 1373, 1380, 1384, 1564–1567, 1650, 1654
\exp_args:Nccc	31, 1650, 1656
\exp_args:Ncco	31, 1687, 1701, 1702
\exp_args:Nccx	31, 1687, 1703
\exp_args:Ncf	30, 1665, 1673, 1958, 1959
\exp_args:NcNc	31, 1687, 1699, 6268

- \exp_args:NcNo 31, 1687, 1700
- \exp_args:Ncnx 31, 1687, 1704
- \exp_args:Nco 30, 1659, 1659, 1665
- \exp_args:Ncx 30, 1665, 1670
- \exp_args:Nf 30, 1661, 1661, 1932, 1933,
1940, 1949, 1982, 1983, 1990, 1998,
2186, 3173, 3175, 3179, 3184, 3187,
3189, 3296, 3410, 3421, 3491, 3502
- \exp_args:Nff .. 30, 1665, 1672, 3298, 3364
- \exp_args:Nfo 30, 1665, 1671
- \exp_args:NNc ... 30, 1120, 1122, 1372,
1379, 1382, 1560–1563, 1650, 1652
- \exp_args:NNf .. 30, 1665, 1665, 1968, 1976
- \exp_args:Nnf 30, 1009, 1015,
1022, 1029, 1665, 1674, 3349, 3353
- \exp_args:Nnnc 31, 1687, 1694
- \exp_args:NNNo 31, 1645, 1648
- \exp_args:NNno 31, 1687, 1689
- \exp_args:Nnno 31, 1687, 1695
- \exp_args:NNNV . 31, 1687, 1687, 4424, 4440
- \exp_args:NNnx 31, 1687, 1690
- \exp_args:Nnnx 31, 1687, 1696
- \exp_args:NNo 30, 1645, 1646, 6429
- \exp_args:Nno 30, 1665, 1675
- \exp_args:NNoo 31, 1687, 1691
- \exp_args:NNox 31, 1687, 1692
- \exp_args:Nnox 31, 1687, 1697
- \exp_args:NNV 30, 1665, 1667
- \exp_args:NNv 30, 1665, 1666
- \exp_args:NnV 30, 1665, 1676
- \exp_args:NNx 30, 1665, 1668, 6292
- \exp_args:Nnx 30, 1665, 1677
- \exp_args:No 29, 1645, 1645, 2196, 4374, 5194
- \exp_args:Noc 30, 1665, 1679
- \exp_args:Noo 30, 1665, 1680
- \exp_args:Nooo 31, 1687, 1707
- \exp_args:Noox 31, 1687, 1706
- \exp_args:Nox 30, 1665, 1681
- \exp_args:NV ... 29, 1661, 1663, 4458, 6705
- \exp_args:Nv 29, 1661, 1662
- \exp_args:NVV 30, 1665, 1683
- \exp_args:Nx 29, 1661, 1664
- \exp_args:Nxo 30, 1665, 1685
- \exp_args:Nxx 30, 1665, 1686
- \exp_eval_error_msg:w 210, 1626, 1629, 1642
- \exp_eval_register:c
210, 1622, 1626, 1636, 1713, 1814, 4622
- \exp_eval_register:N 210, 1614, 1626,
1626, 1637, 1716, 1809, 4619, 4703
- \exp_last_unbraced:NcV
..... 32, 1817, 1821, 2381, 4365
- \exp_last_unbraced:Nf 32, 1817, 1817, 2310
- \exp_last_unbraced:NNNo . 32, 1817, 1833
- \exp_last_unbraced:NNNV . 32, 1817, 1830
- \exp_last_unbraced:NNo
.... 32, 1817, 1827, 5114, 6424, 6723
- \exp_last_unbraced:NNV
..... 32, 1817, 1824, 4663, 6442
- \exp_last_unbraced:No ... 32, 1817, 1819
- \exp_last_unbraced:NV ... 32, 1817, 1818
- \exp_last_unbraced:Nv ... 32, 1817, 1820
- \exp_not:c 31, 1718, 1718,
1739, 1793, 2228, 2232, 2237, 2241,
2244, 2246–2248, 2253, 2255–2257,
2262, 2264–2266, 2271, 2273–2275,
2280, 2282, 2284, 2286, 2288, 2290,
2292, 2294, 2296–2298, 2348, 2357,
2365, 2373, 2377, 2382, 2385, 2389,
2396, 2400, 2405, 2408, 2412, 2951,
6007, 6010, 6013, 6016, 6151, 6156,
6161, 6165, 6254, 6260, 6570, 6831
- \exp_not:f 32, 1708, 1709
- \exp_not:N 15,
31, 878, 879, 1492–1494, 1523–1525,
1575, 1577, 1627, 1718, 2233, 2236,
2240, 2244, 2253, 2262, 2271, 2345,
2347, 2349, 2355, 2363, 2371, 2373,
2377, 2381, 2385, 2389, 2394, 2396,
2400, 2404, 2408, 2412, 2488, 2498,
2627, 2630, 2634, 2638, 2642, 2646,
2650, 2654, 2658, 2662, 2666, 2670,
2678, 2682, 2697, 2918, 2934, 4436,
4446, 4572, 4573, 4577, 4590, 4591,
4859, 6259, 6261, 6407, 6415, 6425,
6571, 6574, 6576, 6728, 6831, 7449,
7492, 7510, 7905, 7942, 8019, 8216,
8322, 8332, 8857, 8870, 8959, 9156,
9169, 9434, 9758, 9767, 9792, 9978,
9980, 9982, 10226, 10228, 10230,
10232, 10234, 10446, 10448, 10450,
10452, 10454, 10456, 10458, 10460
- \exp_not:n 15, 31, 878, 880, 1575,
1578, 1708, 1710, 1713, 1716, 2229,
2350, 2358, 2388, 2411, 2489, 2499,
2949, 2952, 4088, 4094, 4160, 4175,
4199, 4214, 4860, 4949, 5033, 5064,
5155, 5295, 5297, 5316, 5791, 5817,
6258, 6618, 6619, 6845, 10516, 10520

- \exp_not:o 32, 1708, 1708,
4160, 4163, 4166, 4169, 4172, 4175,
4178, 4181, 4184, 4187, 4199, 4202,
4205, 4208, 4211, 4214, 4217, 4220,
4223, 4226, 4779, 4781, 5029, 6488
 - \exp_not:V 32, 1708, 1715, 4163,
4178, 4202, 4217, 5205, 5207, 6621
 - \exp_not:v
32, 1708, 1712, 4166, 4181, 4205, 4220
 - \exp_stop_f: 32, 1600, 1605, 1634, 2311, 4391
 - \expandafter 11, 12, 30,
34, 51, 52, 54, 57, 58, 61, 62, 93, 171
 - \ExplSyntaxNamesOff 6, 682, 686
 - \ExplSyntaxNamesOn 6, 682, 682, 6307
 - \ExplSyntaxOff 4, 6, 96,
648, 651, 763, 770, 781, 802, 810, 6325
 - \ExplSyntaxOn 4, 6,
648, 648, 750, 754, 758, 779, 824, 6303
 - \ExplSyntaxPopStack ... 159, 772, 776, 776
 - \ExplSyntaxStack
.... 159, 762, 769, 772, 776, 777, 784
 - \ExplSyntaxStatus 97, 98,
159, 648, 649, 653, 654, 668, 681, 762
- F**
- \F 2705
 - \fam 163
 - \fi 42, 55, 60, 94, 110, 202, 1275
 - \fi: 8, 848,
860, 864, 966–969, 1050, 1118, 1155,
1177, 1178, 1185, 1191, 1206, 1212,
1291, 1295, 1416, 1423, 1427, 1438,
1475, 1554, 1630, 1633, 1643, 1729,
1754, 1796, 1864, 1867, 1870, 1874,
1877, 1878, 2023, 2052, 2059, 2077,
2078, 2089, 2090, 2105, 2113, 2171,
2484, 2491, 2494, 2501, 2631, 2635,
2639, 2643, 2647, 2651, 2655, 2659,
2663, 2667, 2671, 2675, 2679, 2683,
2690, 2694, 2698, 2734, 2745, 2756,
2767, 2832, 2839, 2842, 2843, 2859–
2864, 2915, 2922, 2938, 3275, 3288,
3516, 3521, 3526, 3530, 3534, 3538,
3542, 3546, 3550, 3555, 3582, 3592,
3602, 3619, 3625, 3626, 3628, 3641,
3645, 3977, 3985, 3989, 3994, 3998,
4002, 4006, 4010, 4014, 4018, 4081,
4143, 4155, 4257, 4264, 4286, 4300,
4569, 4574, 4581, 4592, 5103, 5112,
5125, 5388, 5426, 5429, 5440, 5825
 - \file_add_path:nN
..... 146, 7035, 7035, 7073, 7080
 - \file_add_path_search:nN 7035, 7039, 7045
 - \file_if_exist:n 7071
 - \file_if_exist:nTF 145, 7071
 - \file_input:n 146, 7078, 7078
 - \file_list: 146, 7102, 7102
 - \file_path_include:n ... 146, 7095, 7095
 - \file_path_remove:n 146, 7095, 7100
 - \fileauthor 6, 690, 704, 719
 - \filedate 6, 690, 705,
732, 736, 741, 744, 855, 1571, 1859,
2421, 2513, 3028, 3786, 4052, 4596,
4750, 5075, 5265, 5407, 5559, 5837,
6243, 6352, 6551, 7007, 7119, 10502
 - \filedescription 6, 690,
701, 718, 736, 744, 855, 1571, 1859,
2421, 2513, 3028, 3786, 4052, 4596,
4750, 5075, 5265, 5407, 5559, 5837,
6243, 6352, 6551, 7007, 7119, 10502
 - \filename 6, 690,
702, 716, 736, 744, 855, 1571, 1859,
2421, 2513, 3028, 3786, 4052, 4596,
4750, 5075, 5265, 5407, 5559, 5837,
6243, 6352, 6551, 7007, 7119, 10502
 - \filenameext 6, 690, 706, 733, 740
 - \filetimestamp 6, 690, 707, 734, 742
 - \fileversion 6, 690,
703, 717, 736, 744, 855, 1571, 1859,
2421, 2513, 3028, 3786, 4052, 4596,
4750, 5075, 5265, 5407, 5559, 5837,
6243, 6352, 6551, 7007, 7119, 10502
 - \finalhyphendemerits 351
 - \firstmark 249
 - \firstmarks 475
 - \floatingpenalty 396
 - \font 162
 - \fontchardp 500
 - \fontcharht 499
 - \fontcharic 502
 - \fontcharwd 501
 - \fontdimen 429
 - \fontname 253
 - \fp_abs:c 152, 8004
 - \fp_abs:N 152, 8004, 8004, 8010
 - \fp_abs_aux:NN ... 8004, 8005, 8008, 8012
 - \fp_add:cn 153, 8061
 - \fp_add:Nn 153, 8061, 8061, 8067
 - \fp_add:NNNNNNNN
..... 8452, 8452, 9865, 9913, 9994

\fp_add_aux:NNn .. [8061](#), [8062](#), [8065](#), [8069](#)
\fp_add_core: [8061](#), [8078](#), [8081](#), [8183](#)
\fp_add_difference: [8061](#), [8090](#), [8134](#)
\fp_add_sum: [8061](#), [8088](#), [8118](#)
\fp_compare:NNN [10285](#)
\fp_compare:nNn [10267](#)
\fp_compare:NNNTF [10267](#)
\fp_compare:nNnTF [151](#), [10267](#)
\fp_compare_<: [10267](#)
\fp_compare_<_aux: [10267](#)
\fp_compare_>: [10267](#)
\fp_compare_absolute_a<b: [10267](#)
\fp_compare_absolute_a>b: [10267](#)
\fp_compare_aux:N
..... [10267](#), [10283](#), [10295](#), [10297](#)
\fp_const:cn [148](#), [7411](#)
\fp_const:Nn [148](#), [7411](#), [7411](#), [7422](#)
\fp_cos:cn [155](#), [8921](#)
\fp_cos:Nn [155](#), [8921](#), [8921](#), [8927](#)
\fp_cos_aux:NNn .. [8921](#), [8922](#), [8925](#), [8929](#)
\fp_cos_aux_i: [8921](#), [8954](#), [8964](#)
\fp_cos_aux_ii: .. [8921](#), [8966](#), [8997](#), [9199](#)
\fp_div:cn [153](#), [8298](#)
\fp_div:Nn [153](#), [8298](#), [8298](#), [8304](#)
\fp_div_aux:NNn .. [8298](#), [8299](#), [8302](#), [8306](#)
\fp_div_divide: .. [8298](#), [8389](#), [8405](#), [8429](#)
\fp_div_divide_aux: [8298](#), [8407](#), [8416](#), [8421](#)
\fp_div_integer:NNNN [8591](#),
[8591](#), [9029](#), [9078](#), [9083](#), [9650](#), [10038](#)
\fp_div_internal:
..... [8298](#), [8336](#), [8341](#), [9960](#), [10215](#)
\fp_div_loop: [8298](#), [8346](#), [8387](#), [8401](#), [9247](#)
\fp_div_loop_step:w [8393](#), [8445](#)
\fp_div_store:
.. [8298](#), [8344](#), [8390](#), [8431](#), [8434](#), [9245](#)
\fp_div_store_decimal: . [8298](#), [8434](#), [8436](#)
\fp_div_store_integer:
..... [8298](#), [8344](#), [8432](#), [9245](#)
\fp_exp:cn [154](#), [9395](#)
\fp_exp:Nn [154](#), [9395](#), [9395](#), [9401](#)
\fp_exp_aux: [9395](#), [9453](#), [9463](#), [9473](#)
\fp_exp_aux:NNn .. [9395](#), [9396](#), [9399](#), [9403](#)
\fp_exp_const:cx . [9395](#), [9465](#), [9503](#), [9630](#)
\fp_exp_const:Nx . [9395](#), [9690](#), [9694](#), [10250](#)
\fp_exp_decimal: . [9395](#), [9481](#), [9566](#), [9579](#)
\fp_exp_integer: [9395](#), [9475](#), [9484](#)
\fp_exp_integer_const:n . [9395](#), [9506](#),
[9512](#), [9534](#), [9536](#), [9552](#), [9554](#), [9568](#)
\fp_exp_integer_const:nnnn
..... [9395](#), [9570](#), [9573](#), [9950](#)
\fp_exp_integer_tens:
..... [9395](#), [9490](#), [9509](#), [9514](#), [9518](#)
\fp_exp_integer_units: . [9395](#), [9547](#), [9549](#)
\fp_exp_internal: [9395](#), [9429](#), [9446](#), [10251](#)
\fp_exp_overflow_msg:
..... [9457](#), [9470](#), [10475](#), [10481](#)
\fp_exp_Taylor: .. [9395](#), [9608](#), [9643](#), [9687](#)
\fp_extended_normalise:
. [8607](#), [8607](#), [8709](#), [9448](#), [10125](#), [10159](#)
\fp_extended_normalise_aux:NNNNNNNN [8607](#)
\fp_extended_normalise_aux_i:
..... [8607](#), [8608](#), [8611](#), [8617](#)
\fp_extended_normalise_aux_i:w
..... [8607](#), [8615](#), [8621](#)
\fp_extended_normalise_aux_ii:
..... [8607](#), [8609](#), [8636](#), [8642](#)
\fp_extended_normalise_aux_ii:w
..... [8607](#), [8626](#), [8630](#)
\fp_extended_normalise_ii_aux:NNNNNNNN [8640](#), [8646](#)
\fp_extended_normalise_output:
..... [8667](#), [8667](#), [8673](#), [9545](#), [9564](#)
\fp_extended_normalise_output_aux:N
..... [8667](#), [8692](#), [8694](#)
\fp_extended_normalise_output_aux_i:NNNNNNNN
..... [8667](#), [8671](#), [8677](#)
\fp_extended_normalise_output_aux_ii:NNNNNNNN
..... [8667](#), [8686](#), [8690](#)
\fp_gabs:c [152](#), [8004](#)
\fp_gabs:N [152](#), [8004](#), [8007](#), [8011](#)
\fp_gadd:cn [153](#), [8061](#)
\fp_gadd:Nn [153](#), [8061](#), [8064](#), [8068](#)
\fp_gcos:cn [155](#), [8921](#)
\fp_gcos:Nn [155](#), [8921](#), [8924](#), [8928](#)
\fp_gdiv:cn [153](#), [8298](#)
\fp_gdiv:Nn [153](#), [8298](#), [8301](#), [8305](#)
\fp_gexp:cn [154](#), [9395](#)
\fp_gexp:Nn [154](#), [9395](#), [9398](#), [9402](#)
\fp_gln:cn [154](#), [9731](#)
\fp_gln:Nn [154](#), [9731](#), [9734](#), [9738](#)
\fp_gmul:cn [153](#), [8186](#)
\fp_gmul:Nn [153](#), [8186](#), [8189](#), [8193](#)
\fp_gneg:c [152](#), [8032](#)
\fp_gneg:N [152](#), [8032](#), [8035](#), [8039](#)
\fp_gpow:cn [154](#), [10059](#)
\fp_gpow:Nn [154](#), [10059](#), [10062](#), [10066](#)
\fp_ground_figures:cn [151](#), [7886](#)
\fp_ground_figures:Nn [151](#), [7886](#), [7890](#), [7893](#)
\fp_ground_places:cn [151](#), [7922](#)

\fp_ground_places:Nn [151](#), [7922](#), [7926](#), [7929](#)
\fp_gset:c [138](#)
\fp_gset:cn [149](#), [7431](#)
\fp_gset:N [138](#)
\fp_gset:Nn ... [149](#), [7415](#), [7431](#), [7434](#), [7467](#)
\fp_gset_eq:cc [148](#), [7518](#), [7525](#)
\fp_gset_eq:cN [148](#), [7518](#), [7523](#)
\fp_gset_eq:Nc [148](#), [7518](#), [7524](#)
\fp_gset_eq:NN [148](#), [7518](#), [7522](#)
\fp_gset_from_dim:cn [149](#), [7468](#)
\fp_gset_from_dim:Nn [149](#), [7468](#), [7471](#), [7515](#)
\fp_gsin:cn [155](#), [8822](#)
\fp_gsin:Nn [155](#), [8822](#), [8825](#), [8829](#)
\fp_gsub:cn [153](#), [8165](#)
\fp_gsub:Nn [153](#), [8165](#), [8168](#), [8172](#)
\fp_gtan:cn [155](#), [9121](#)
\fp_gtan:Nn [155](#), [9121](#), [9124](#), [9128](#)
\fp_gzero:c [148](#), [7423](#)
\fp_gzero:N [148](#), [7423](#), [7426](#), [7430](#)
\fp_if_undefined:N [10253](#)
\fp_if_undefined:NTF [151](#), [10253](#)
\fp_if_undefined_p:N [151](#), [10253](#)
\fp_if_zero:N [10260](#)
\fp_if_zero:NTF [151](#), [10260](#)
\fp_if_zero_p:N [151](#), [10260](#)
\fp_level_input_exponents:
..... [7352](#), [7352](#), [8082](#)
\fp_level_input_exponents_a:
..... [7352](#), [7354](#), [7359](#), [7365](#)
\fp_level_input_exponents_a:NNNNNNNN
..... [7352](#), [7363](#), [7369](#)
\fp_level_input_exponents_b:
..... [7352](#), [7356](#), [7382](#), [7388](#)
\fp_level_input_exponents_b:NNNNNNNN
..... [7352](#), [7386](#), [7392](#)
\fp_ln:cn [154](#), [9731](#)
\fp_ln:Nn [154](#), [9731](#), [9731](#), [9737](#)
\fp_ln_aux: [9731](#), [9753](#), [9773](#)
\fp_ln_aux:NNn ... [9731](#), [9732](#), [9735](#), [9739](#)
\fp_ln_const:nn .. [9847](#), [9863](#), [9910](#), [9948](#)
\fp_ln_error_msg: [9761](#), [9769](#), [10484](#), [10487](#)
\fp_ln_exponent: [9731](#), [9787](#), [9796](#)
\fp_ln_exponent_units: . [9731](#), [9859](#), [9861](#)
\fp_ln_fixed: ... [9731](#), [9961](#), [10004](#), [10010](#)
\fp_ln_fixed_aux:NNNNNNNN
..... [9731](#), [10008](#), [10014](#)
\fp_ln_integer_const:nn [9731](#)
\fp_ln_internal: . [9731](#), [9797](#), [9830](#), [10222](#)
\fp_ln_mantissa: [9731](#), [9872](#), [9905](#)
\fp_ln_mantissa_aux: [9731](#), [9908](#), [9930](#), [9934](#)
\fp_ln_mantissa_divide_two:
..... [9731](#), [9933](#), [9937](#)
\fp_ln_normalise:
..... [9731](#), [9864](#), [9874](#), [9880](#), [9911](#)
\fp_ln_normalise_aux:NNNNNNNN
..... [9878](#), [9884](#)
\fp_ln_normalise_aux:NNNNNNNN .. [9731](#)
\fp_ln_Taylor: [9731](#), [9927](#), [9953](#)
\fp_ln_Taylor_aux: [9731](#), [9974](#), [10028](#), [10056](#)
\fp_mul:cn [153](#), [8186](#)
\fp_mul:Nn [153](#), [8186](#), [8186](#), [8192](#)
\fp_mul:NNNNNN [8489](#), [8489](#),
..... [9025](#), [9070](#), [9074](#), [9646](#), [9969](#), [10030](#)
\fp_mul:NNNNNNNN
..... [8532](#), [8532](#), [9538](#), [9557](#), [9609](#), [10239](#)
\fp_mul_aux:NNn .. [8186](#), [8187](#), [8190](#), [8194](#)
\fp_mul_end_level: [8186](#),
..... [8259](#), [8263](#), [8266](#), [8270](#), [8288](#), [8512](#),
..... [8517](#), [8521](#), [8526](#), [8528](#), [8529](#), [8557](#),
..... [8564](#), [8570](#), [8577](#), [8581](#), [8584](#), [8588](#)
\fp_mul_end_level:NNNNNNNN
..... [8186](#), [8291](#), [8293](#)
\fp_mul_internal: [8186](#), [8203](#), [8244](#)
\fp_mul_product:NN
..... [8251](#)–[8253](#), [8255](#)–[8258](#), [8260](#)–[8262](#),
..... [8264](#), [8265](#), [8269](#), [8284](#), [8500](#)–[8505](#),
..... [8507](#)–[8511](#), [8513](#)–[8516](#), [8518](#)–[8520](#),
..... [8524](#), [8525](#), [8527](#), [8543](#)–[8548](#), [8550](#)–
..... [8556](#), [8558](#)–[8563](#), [8565](#)–[8569](#), [8573](#)–
..... [8576](#), [8578](#)–[8580](#), [8582](#), [8583](#), [8587](#)
\fp_mul_split:NNNN
..... [8186](#), [8245](#), [8247](#), [8273](#), [8490](#), [8492](#),
..... [8494](#), [8496](#), [8533](#), [8535](#), [8537](#), [8539](#)
\fp_mul_split:w [8186](#)
\fp_mul_split_aux:w [8275](#), [8281](#)
\fp_neg:c [152](#), [8032](#)
\fp_neg:N [152](#), [8032](#), [8032](#), [8038](#)
\fp_neg:NN [8032](#)
\fp_neg_aux:NN [8033](#), [8036](#), [8040](#)
\fp_new:c [148](#), [7406](#)
\fp_new:N [148](#), [7406](#), [7406](#), [7410](#), [7414](#)
\fp_overflow_msg: [7282](#), [7347](#), [10466](#), [10472](#)
\fp_pow:cn [154](#), [10059](#)
\fp_pow:Nn [154](#), [10059](#), [10059](#), [10065](#)
\fp_pow_aux:NNn [10059](#), [10060](#), [10063](#), [10067](#)
\fp_pow_aux_i: [10059](#), [10112](#), [10117](#)
\fp_pow_aux_ii: [10059](#), [10120](#), [10134](#), [10152](#)
\fp_pow_aux_iii: ... [10059](#), [10176](#), [10206](#)
\fp_pow_aux_iv: [10059](#), [10154](#),
..... [10168](#), [10182](#), [10186](#), [10208](#), [10217](#)

\fp_pow_negative: [10059](#)
 \fp_pow_positive: [10059](#)
 \fp_read:N [7200](#),
 [7200](#), [7896](#), [7932](#), [8014](#), [8042](#), [8071](#),
 [8175](#), [8196](#), [8308](#), [10069](#), [10289](#), [10294](#)
 \fp_read_aux:w [7200](#), [7201](#), [7203](#)
 \fp_round: [7899](#), [7936](#), [7959](#), [7959](#)
 \fp_round_aux:NNNNNNNN [7959](#), [7965](#), [7967](#)
 \fp_round_figures:cn [150](#), [7886](#)
 \fp_round_figures:Nn [150](#), [7886](#), [7886](#), [7889](#)
 \fp_round_figures_aux:NNn
 [7886](#), [7887](#), [7891](#), [7894](#)
 \fp_round_loop:N . [7959](#), [7968](#), [7979](#), [8001](#)
 \fp_round_places:cn [151](#), [7922](#)
 \fp_round_places_aux:Nn [151](#), [7922](#), [7922](#), [7925](#)
 \fp_round_places_aux:NNn
 [7922](#), [7923](#), [7927](#), [7930](#)
 .fp_set:c [138](#)
 \fp_set:cn [148](#), [7431](#)
 .fp_set:N [138](#)
 \fp_set:Nn [148](#), [7431](#), [7431](#), [7466](#)
 \fp_set_aux:NNn .. [7431](#), [7432](#), [7435](#), [7437](#)
 \fp_set_eq:cc [148](#), [7518](#), [7521](#)
 \fp_set_eq:cN [148](#), [7518](#), [7519](#)
 \fp_set_eq:Nc [148](#), [7518](#), [7520](#)
 \fp_set_eq:NN [148](#), [7518](#), [7518](#)
 \fp_set_from_dim:cn [149](#), [7468](#)
 \fp_set_from_dim:Nn [149](#), [7468](#), [7468](#), [7514](#)
 \fp_set_from_dim_aux:NNn
 [7468](#), [7469](#), [7472](#), [7474](#)
 \fp_set_from_dim_aux:w
 [7468](#), [7480](#), [7509](#), [7510](#), [7513](#)
 \fp_show:c [149](#), [7526](#), [7527](#)
 \fp_show:N [149](#), [7526](#), [7526](#)
 \fp_sin:cn [155](#), [8822](#)
 \fp_sin:Nn [155](#), [8822](#), [8822](#), [8828](#)
 \fp_sin_aux:NNn .. [8822](#), [8823](#), [8826](#), [8830](#)
 \fp_sin_aux_i: [8822](#), [8865](#), [8876](#)
 \fp_sin_aux_ii: .. [8822](#), [8878](#), [8909](#), [9221](#)
 \fp_split:Nn .. [7213](#), [7213](#), [7439](#), [7478](#),
 [8072](#), [8176](#), [8197](#), [8309](#), [8832](#), [8931](#),
 [9131](#), [9405](#), [9741](#), [10074](#), [10271](#), [10277](#)
 \fp_split_aux_i:w [7213](#), [7251](#), [7255](#)
 \fp_split_aux_ii:w [7213](#), [7256](#), [7258](#)
 \fp_split_aux_iii:w [7213](#), [7259](#), [7261](#)
 \fp_split_decimal:w [7213](#), [7263](#), [7266](#)
 \fp_split_decimal_aux:w [7213](#), [7267](#), [7269](#)
 \fp_split_exponent: [7213](#)
 \fp_split_exponent:w [7220](#), [7248](#)
 \fp_split_sign: [7213](#), [7218](#), [7222](#), [7232](#), [7243](#)
 \fp_standardise:NNNN [7285](#), [7285](#),
 [7440](#), [7483](#), [8073](#), [8093](#), [8177](#), [8198](#),
 [8208](#), [8310](#), [8351](#), [8833](#), [8886](#), [8932](#),
 [8974](#), [9132](#), [9216](#), [9225](#), [9252](#), [9406](#),
 [9625](#), [9742](#), [9807](#), [10075](#), [10272](#), [10278](#)
 \fp_standardise_aux: [7285](#), [7297](#),
 [7303](#), [7313](#), [7314](#), [7320](#), [7337](#), [7350](#)
 \fp_standardise_aux:NNNN [7285](#), [7292](#), [7296](#)
 \fp_standardise_aux:w
 .. [7285](#), [7301](#), [7307](#), [7319](#), [7324](#), [7351](#)
 \fp_sub:cn [153](#), [8165](#)
 \fp_sub:Nn [153](#), [8165](#), [8165](#), [8171](#)
 \fp_sub:NNNNNNNN [8467](#),
 [8467](#), [8788](#), [8798](#), [8809](#), [9915](#), [9996](#)
 \fp_sub_aux:NNn .. [8165](#), [8166](#), [8169](#), [8173](#)
 \fp_tan:cn [155](#), [9121](#)
 \fp_tan:Nn [155](#), [9121](#), [9121](#), [9127](#)
 \fp_tan_aux:NNn .. [9121](#), [9122](#), [9125](#), [9129](#)
 \fp_tan_aux_i: [9121](#), [9164](#), [9175](#)
 \fp_tan_aux_ii: [9121](#), [9177](#), [9184](#)
 \fp_tan_aux_iii: . [9121](#), [9206](#), [9209](#), [9212](#)
 \fp_tan_aux_iv: .. [9121](#), [9236](#), [9239](#), [9242](#)
 \fp_tmp:w
 [7405](#), [7405](#), [7446](#), [7464](#), [7489](#), [7507](#),
 [7902](#), [7920](#), [7939](#), [7957](#), [8016](#), [8030](#),
 [8079](#), [8098](#), [8184](#), [8213](#), [8242](#), [8319](#),
 [8329](#), [8339](#), [8356](#), [8854](#), [8867](#), [8874](#),
 [8956](#), [8962](#), [9153](#), [9166](#), [9173](#), [9431](#),
 [9444](#), [9755](#), [9764](#), [9771](#), [9789](#), [9975](#),
 [9985](#), [10088](#), [10094](#), [10105](#), [10115](#),
 [10137](#), [10144](#), [10189](#), [10223](#), [10237](#)
 \fp_to_dim:c [150](#), [7595](#)
 \fp_to_dim:N [150](#), [7595](#), [7595](#), [7596](#)
 \fp_to_int:c [150](#), [7597](#)
 \fp_to_int:N [150](#), [7597](#), [7597](#), [7600](#)
 \fp_to_int_aux:w [7597](#), [7598](#), [7601](#)
 \fp_to_int_large:w [7597](#), [7608](#), [7622](#)
 \fp_to_int_large_aux:nnn
 [7597](#), [7634](#), [7637](#), [7640](#),
 [7643](#), [7646](#), [7649](#), [7653](#), [7657](#), [7660](#)
 \fp_to_int_large_aux_1:w [7597](#)
 \fp_to_int_large_aux_2:w [7597](#)
 \fp_to_int_large_aux_3:w [7597](#)
 \fp_to_int_large_aux_4:w [7597](#)
 \fp_to_int_large_aux_5:w [7597](#)
 \fp_to_int_large_aux_6:w [7597](#)
 \fp_to_int_large_aux_7:w [7597](#)
 \fp_to_int_large_aux_8:w [7597](#)
 \fp_to_int_large_aux_i:w [7597](#), [7624](#), [7630](#)
 \fp_to_int_large_aux_ii:w [7597](#), [7626](#), [7667](#)

\fp_to_int_none:w	7597	\fp_use_i_to_vii:NNNNNNNNN	
\fp_to_int_small:w	7597 , 7606 , 7612	7671 , 7780 , 7786 , 7880
\fp_to_tl:c	150 , 7671	\fp_use_iix_ix:NNNNNNNNN	7671 , 7778 , 7878
\fp_to_tl:N	150 , 7671 , 7671 , 7674	\fp_use_ix:NNNNNNNNN ...	7671 , 7759 , 7879
\fp_to_tl_aux:w	7671 , 7672 , 7675	\fp_use_large:w	7528 , 7537 , 7555
\fp_to_tl_large:w	7671 , 7682 , 7686	\fp_use_large_aux_1:w	7528
\fp_to_tl_large_0:w	7671	\fp_use_large_aux_2:w	7528
\fp_to_tl_large_1:w	7671	\fp_use_large_aux_3:w	7528
\fp_to_tl_large_2:w	7671	\fp_use_large_aux_4:w	7528
\fp_to_tl_large_3:w	7671	\fp_use_large_aux_5:w	7528
\fp_to_tl_large_4:w	7671	\fp_use_large_aux_6:w	7528
\fp_to_tl_large_5:w	7671	\fp_use_large_aux_7:w	7528
\fp_to_tl_large_6:w	7671	\fp_use_large_aux_8:w	7528
\fp_to_tl_large_7:w	7671	\fp_use_large_aux_i:w ..	7528 , 7557 , 7563
\fp_to_tl_large_8:w	7671	\fp_use_large_aux_ii:w ..	7528 , 7559 , 7590
\fp_to_tl_large_8_aux:w	7671	\fp_use_none:w	7528 , 7544 , 7549
\fp_to_tl_large_9:w	7671	\fp_use_small:w	7528 , 7541 , 7550
\fp_to_tl_large_aux_i:w	7671 , 7688 , 7694	\fp_zero:c	148 , 7423
\fp_to_tl_large_aux_ii:w	7671 , 7690 , 7697	\fp_zero:N	148 , 7423 , 7423 , 7429
\fp_to_tl_large_zeros:NNNNNNNNN	7671 , 7699 , 7704 , 7708 , 7712 , 7716 , 7720 , 7724 , 7728 , 7732 , 7741 , 7799 , 7802	\frozen@everydisplay	789
\fp_to_tl_small:w	7671 , 7680 , 7744	\frozen@everymath	788
\fp_to_tl_small_aux:w ..	7671 , 7753 , 7797	\futurelet	158
\fp_to_tl_small_one:w ..	7671 , 7746 , 7758		
\fp_to_tl_small_two:w ..	7671 , 7750 , 7777		
\fp_to_tl_small_zeros:NNNNNNNNN 7671 , 7764 , 7774 , 7783 , 7794 , 7840		
\fp_trig_calc_cos:			
.....	8914 , 8916 , 9000 , 9006 , 9019 , 9019		
\fp_trig_calc_sin:			
.....	8912 , 8918 , 9002 , 9004 , 9019 , 9054		
\fp_trig_calc_Taylor:			
.....	9019 , 9051 , 9065 , 9068 , 9118		
\fp_trig_normalise:			
.....	8706 , 8706 , 8877 , 8965 , 9185		
\fp_trig_normalise_aux:			
.....	8706 , 8710 , 8724 , 8728 , 8736		
\fp_trig_octant:	8715 , 8779 , 8779		
\fp_trig_octant_aux:			
.....	8779 , 8781 , 8796 , 8805 , 8818		
\fp_trig_overflow_msg:			
.....	8721 , 9181 , 10490 , 10496		
\fp_trig_sub:NNN ..	8706 , 8726 , 8732 , 8741		
\fp_use:c	149 , 7528		
\fp_use:N	149 , 7528 , 7528 , 7531 , 7595		
\fp_use_aux:w	7528 , 7529 , 7532		
\fp_use_i_to_iix:NNNNNNNNN			
.....	7671 , 7761 , 7767 , 7883		

- \g_iow_tmp_stream 123, 5665, 5674, 5675, 5678
 - \g_peek_token 53, 2866, 2867, 2871
 - \g_prg_inline_level_int 1857, 1954, 1955, 1956, 1960, 1962, 3737
 - \g_prop_inline_level_int 111, 5393, 5393, 5395, 5396, 5399, 5400
 - \g_seq_nesting_depth_int 4825, 4825, 4838, 4840, 4844, 4846
 - \g_tl_inline_level_int 83, 3736, 4352, 4353, 4354, 4357, 4359, 4362, 4363, 4366, 4368
 - \g_tmpa_bool 222, 2020, 2021
 - \g_tmpa_dim 75, 3960, 3964
 - \g_tmpa_int 65, 3290, 3293
 - \g_tmpa_skip 71, 3875, 3880
 - \g_tmpa_tl 83, 4247, 4247
 - \g_tmpa_toks 90, 4737, 4741
 - \g_tmpb_dim 75, 3960, 3965
 - \g_tmpb_int 65, 3290, 3294
 - \g_tmpb_skip 71, 3875, 3881
 - \g_tmpb_tl 83, 4247, 4248
 - \g_tmpb_toks 90, 4737, 4742
 - \g_tmppc_toks 90, 4737, 4743
 - \g_toks_allocation_seq 4738
 - \g_trace_statistics_status 841
 - \g_xref_all_curr_deferred_fields_prop 380, 6247, 6248, 6295
 - \g_xref_all_curr_immediate_fields_prop 380, 6247, 6247, 6291
 - \gdef 149
 - .generate_choices:n 138
 - \GetIdInfo 6, 690, 690
 - \GetIdInfoAuxCVS:w 690, 725, 730
 - \GetIdInfoAuxi:w 690, 710, 714
 - \GetIdInfoAuxii:w 690, 720, 723
 - \GetIdInfoAuxSVN:w 690, 727, 738
 - \GetIdInfoMaybeMissing:w 693, 695
 - \getname 6315, 6343, 6346
 - \getpage 6318, 6344, 6347
 - \getvaluepage 6321, 6344, 6347
 - \global 164
 - \globaldefs 168
 - \glueexpr 508
 - \glueshrink 511
 - \glueshrinkorder 513
 - \gluestretch 510
 - \gluestretchorder 512
 - \gluetomu 514
 - \group_align_safe_begin: 40, 1876, 1876, 2046, 2882, 2897
 - \group_align_safe_end: 40, 1876, 1878, 2141, 2142, 2880, 2881, 2896, 2907
 - \group_begin: 25, 891, 892, 940, 1141, 1639, 1756, 2304, 2317, 2614, 2703, 2795, 4271, 4409, 4419, 4435, 4445, 4453, 4978, 5812, 5876, 5927, 5936, 5971, 5978, 6280, 6306, 6391, 6404, 6437, 6448, 7438, 7475, 7895, 7931, 8013, 8041, 8070, 8174, 8195, 8307, 8831, 8930, 9130, 9404, 9740, 9954, 10068, 10124, 10157, 10218, 10270, 10288, 10523, 10595
 - \group_end: 25, 891, 893, 943, 1145, 1644, 1767, 2308, 2321, 2628, 2713, 2803, 4275, 4278, 4415, 4425, 4441, 4450, 4458, 4989, 5812, 5879, 5934, 5955, 5975, 5982, 6288, 6309, 6396, 6434, 6444, 6451, 7448, 7491, 7904, 7941, 8018, 8058, 8100, 8215, 8321, 8331, 8358, 8856, 8869, 8958, 9155, 9168, 9433, 9757, 9766, 9791, 9977, 10090, 10096, 10107, 10130, 10136, 10139, 10146, 10162, 10170, 10179, 10191, 10225, 10301, 10311, 10314, 10318, 10322, 10326, 10337, 10341, 10344, 10353, 10356, 10366, 10370, 10383, 10387, 10391, 10396, 10401, 10404, 10414, 10418, 10422, 10427, 10432, 10435, 10529, 10598
 - \group_execute_after:N 25, 894, 894
- ## H
- \H 2710
 - \halign 175
 - \hangafter 353
 - \hangindent 354
 - \hbadness 415
 - \hbox 410
 - \hbox:n 116, 5527, 5527
 - \hbox:w 849
 - \hbox_gset:cn 116, 5528
 - \hbox_gset:Nn 116, 5528, 5530, 5531
 - \hbox_gset_inline_begin:c 116, 5538
 - \hbox_gset_inline_begin:N 116, 5538, 5543, 5546
 - \hbox_gset_inline_end: . 116, 5538, 5547
 - \hbox_gset_to_wd:cn 116, 5532
 - \hbox_gset_to_wd:Nnn 116, 5532, 5536, 5537

<code>\hbox_overlap_left:n</code> . . .	116, 5550, 5550	<code>\if_hbox:N</code>	112, 5422, 5422
<code>\hbox_overlap_right:n</code> . .	116, 5550, 5551	<code>\if_horizontal_mode:</code>	843
<code>\hbox_set:cn</code>	116, 5528	<code>\if_int_compare:w</code>	67, 3032, 3035, 3525, 3529, 3533, 3537, 3541, 3545, 3549, 3553, 3580, 3587, 3597, 3610, 3614, 3615, 3621, 3741
<code>\hbox_set:Nn</code>	116, 5528, 5528–5530	<code>\if_int_odd:w</code>	68, 3032, 3037, 3640, 3644, 3742
<code>\hbox_set_inline_begin:c</code>	116, 5538	<code>\if_intexpr_case:w</code>	3743
<code>\hbox_set_inline_begin:N</code>	116, 5538, 5538, 5541, 5544	<code>\if_intexpr_compare:w</code>	3741
<code>\hbox_set_inline_end:</code> . .	116, 5538, 5542	<code>\if_intexpr_odd:w</code>	3742
<code>\hbox_set_to_wd:cnn</code>	116, 5532	<code>\if_meaning:w</code>	9, 871, 871, 1151, 1170, 1187, 1553, 1627, 1628, 1794, 2052, 2059, 2070, 2073, 2082, 2085, 2101, 2109, 2483, 2494, 2674, 2697, 2729, 2740, 2751, 2762, 2911, 3521, 3989, 4076, 4256, 4264, 4285, 4300, 4568, 5100, 5111, 5120, 5386
<code>\hbox_set_to_wd:Nnn</code>	116, 5532, 5532, 5535, 5536	<code>\if_mode_horizontal:</code> . .	9, 872, 873, 1867
<code>\hbox_to_wd:nn</code>	116, 5548, 5548	<code>\if_mode_inner:</code>	9, 872, 875, 1870
<code>\hbox_to_zero:n</code> . . .	116, 5548, 5549–5551	<code>\if_mode_math:</code>	9, 872, 872, 1873
<code>\hbox_unpack:c</code>	117, 5552	<code>\if_mode_vertical:</code>	9, 872, 874, 1864
<code>\hbox_unpack:N</code>	117, 5552, 5552, 5553	<code>\if_num:w</code>	67, 1877, 1878, 2487, 2497, 3032, 3036
<code>\hbox_unpack_clear:c</code>	117, 5552	<code>\if_predicate:w</code>	9, 860, 868, 1434, 2170, 2689, 2693, 2827, 2828, 2834, 2835, 2846, 2848, 2850, 2852, 2854, 2856, 4139, 4151, 4584
<code>\hbox_unpack_clear:N</code>	117, 5552, 5554, 5555	<code>\if_true:</code>	8, 860, 860
<code>\hfil</code>	318	<code>\if_vbox:N</code>	112, 5422, 5423
<code>\hfill</code>	320	<code>\ifcase</code>	185
<code>\hfilneg</code>	319	<code>\ifcat</code>	186
<code>\hfuzz</code>	417	<code>\ifcsname</code>	469
<code>\hoffset</code>	392	<code>\ifdefined</code>	468
<code>\holdinginserts</code>	395	<code>\ifdim</code>	189
<code>\hrule</code>	331	<code>\ifeof</code>	190
<code>\hsize</code>	356	<code>\iffalse</code>	195
<code>\hskip</code>	321	<code>\iffontchar</code>	498
<code>\hss</code>	322	<code>\ifhbox</code>	191
<code>\ht</code>	460	<code>\ifhmode</code>	197
<code>\hyphenation</code>	446	<code>\ifinner</code>	200
<code>\hyphenchar</code>	430	<code>\ifmmode</code>	198
<code>\hyphenpenalty</code>	348	<code>\ifnum</code>	187
I			
<code>\I</code>	2710	<code>\ifodd</code>	97, 188
<code>\if</code>	184	<code>\iftrue</code>	196
<code>\if:w</code>	9, 860, 866, 1048, 1116, 1727, 1754, 3516, 3985, 4572	<code>\ifvbox</code>	192
<code>\if_bool:N</code>	9, 860, 867, 1415, 1422, 1426, 2023	<code>\ifvmode</code>	199
<code>\if_box_empty:N</code>	112, 5422, 5424	<code>\ifvoid</code>	193
<code>\if_case:w</code>	67, 3032, 3038, 3265, 3278, 3743	<code>\ifx</code>	12, 52, 58, 62, 194
<code>\if_catcode:w</code>	9, 860, 870, 2630, 2634, 2638, 2642, 2646, 2650, 2654, 2658, 2662, 2666, 2670, 2678, 2918, 4590	<code>\ignorespaces</code>	242
<code>\if_charcode:w</code>	9, 860, 869, 2682, 2934, 4577		
<code>\if_cs_exist:N</code>	9, 876, 876, 1173		
<code>\if_cs_exist:w</code>	9, 876, 877, 1181		
<code>\if_dim:w</code>	73, 3971, 3971, 3976, 3993, 3997, 4001, 4005, 4009, 4013, 4017		
<code>\if_eof:w</code>	122, 5822, 5822		
<code>\if_false:</code>	8, 860, 861, 1877		

- \immediate 204
- \indent 338
- \initcatcodetable 638
- \input 212
- \input@path 7049, 7052, 7066
- \InputIfFileExists 6308
- \inputlineno 214
- \insert 394
- \insertpenalties 397
- \int_abs:n 56, 3578, 3578, 3756
- \int_add:cn 57, 3113
- \int_add:Nn 57, 3101, 3102, 3113, 3113, 3130, 3137
- \int_advance:w 266, 3032, 3040, 3079, 3084, 3114, 3120
- \int_compare:n 3510
- \int_compare:nF 3651, 3657, 3749, 5748, 5760
- \int_compare:nNn 3552
- \int_compare:nNnF 1882, 1884, 1937, 1946, 1965, 1973, 1987, 1995, 3564, 3663, 3669, 3754
- \int_compare:nNnT 1880, 3557, 3660, 3666, 3753, 10569
- \int_compare:nNnTF 59, 1931, 1957, 1981, 2166, 2191, 3171, 3182, 3325, 3346, 3358, 3427, 3431, 3508, 3552, 3571, 3752, 10538
- \int_compare:nT 3648, 3654, 3748, 5672, 5692, 5712, 5732
- \int_compare:nTF 59, 3510, 3672, 3674, 3747, 5629, 5642, 6678
- \int_compare_auxi:w 3511, 3514
- \int_compare_auxii:w 3515, 3518
- \int_compare_p:n 59, 3510, 3746
- \int_compare_p:nNn 59, 3552, 3751, 3894, 3895
- \int_const:cn 65, 3671, 3722–3735
- \int_const:Nn ... 65, 3671, 3671, 3690, 3693–3708, 3710–3721, 7123–7129
- \int_convert_from_base_ten:nn 63, 3345, 3345, 3479, 3482, 3485
- \int_convert_from_base_ten_aux:nnn 3345, 3349, 3353, 3357, 3364
- \int_convert_number_to_letter:n 266, 3345, 3360, 3366, 3374
- \int_convert_number_with_rule:nnN ... 67, 3181, 3181, 3184, 3256, 3260
- \int_convert_to_base_ten:nn 63, 3406, 3406, 3470, 3473, 3476
- \int_convert_to_base_ten_aux:N 3406, 3422, 3426
- \int_convert_to_base_ten_aux:nn 3406, 3410, 3414
- \int_convert_to_base_ten_aux:nnN ... 3406, 3415, 3417, 3421
- \int_convert_to_symbols:nnn 63, 3170, 3170, 3173, 3192, 3223
- \int_decr:c 57, 3079
- \int_decr:N 57, 3079, 3084, 3100, 3102, 3106, 6385
- \int_div_round:nn .. 56, 3605, 3632, 3761
- \int_div_truncate:nn 56, 3174, 3185, 3370, 3605, 3608, 3636, 3760
- \int_div_truncate_raw:nn 3605
- \int_do_until:nn 60, 3647, 3656, 3657, 3777
- \int_do_until:nNnn 60, 3659, 3668, 3669, 3782
- \int_do_while:nn ... 60, 3647, 3653, 3776
- \int_do_while:nNnn 60, 3654, 3659, 3665, 3666, 3781
- \int_eval:n 55, 1892, 1934, 1941, 1950, 1961, 1969, 1977, 1984, 1991, 1999, 2186, 3032, 3041, 3041, 3145, 3160, 3164, 3176, 3179, 3187, 3189, 3296, 3298, 3350, 3354, 3407, 3422, 3430, 3488, 3503, 3507, 3606, 3632, 3744, 4394, 4400
- \int_eval:w 68, 2519, 2523, 2529, 2565, 2569, 2573, 2577, 2581, 2585, 2588, 2591, 2595, 2598, 2601, 2605, 2609, 3032, 3033, 3042, 3054, 3114, 3120, 3512, 3525, 3529, 3533, 3537, 3541, 3545, 3549, 3553, 3580, 3583, 3586, 3588, 3596, 3598, 3609, 3610, 3614, 3615, 3621, 3635, 3640, 3644, 3739
- \int_eval_end: 68, 2519, 2523, 2529, 2565, 2569, 2573, 2577, 2581, 2585, 2588, 2591, 2595, 2598, 2601, 2605, 2609, 3032, 3034, 3042, 3054, 3114, 3120, 3525, 3529, 3533, 3537, 3541, 3545, 3549, 3554, 3583, 3588, 3593, 3598, 3603, 3630, 3637, 3640, 3644, 3740
- \int_from_alph:n 62, 3487, 3487
- \int_from_alph_aux:N ... 3487, 3503, 3506
- \int_from_alph_aux:n ... 3487, 3491, 3495
- \int_from_alph_aux:nN 3487, 3496, 3498, 3502
- \int_from_binary:n 62, 3469, 3469

`\int_from_hexadecimal:n` . 63, 3469, 3472
`\int_from_octal:n` 63, 3469, 3475
`\int_from_roman:n` 63, 3300, 3300
`\int_from_roman_aux:NN`
. 3300, 3305, 3308, 3332, 3336
`\int_from_roman_clean_up:w`
. 3300, 3315, 3322, 3324, 3344
`\int_from_roman_end:w` . . 3300, 3303, 3341
`\int_gadd:cn` 57, 3113
`\int_gadd:Nn` 57, 3103,
3104, 3113, 3125, 3138, 10537, 10568
`\int_gdecr:c` 57, 3079
`\int_gdecr:N`
. 57, 1962, 2389, 2412, 3079, 3095,
3104, 3108, 4359, 4368, 4844, 5400
`\int_get_digits:n`
. 266, 3406, 3411, 3442, 3492
`\int_get_sign:n` 266, 3406, 3409, 3438, 3490
`\int_get_sign_and_digits:n`
. 266, 3406, 3434
`\int_get_sign_and_digits_aux:nNNN` . .
. . 3406, 3435, 3439, 3443, 3446, 3468
`\int_get_sign_and_digits_aux:oNNN` . .
. 3406, 3451, 3455, 3461
`\int_gincr:c` 57, 3079
`\int_gincr:N`
. 57, 1955, 2373, 2396, 3079, 3089,
3103, 3107, 4353, 4362, 4840, 5395
`.int_gset:c` 138
`\int_gset:cn` 58, 3054
`.int_gset:N` 138
`\int_gset:Nn`
58, 3054, 3059, 3066, 3074, 3677, 3687
`\int_gset_eq:cc` 57, 3067
`\int_gset_eq:cN` 57, 3067
`\int_gset_eq:Nc` 57, 3067
`\int_gset_eq:NN` . 57, 3067, 3067, 3070–3072
`\int_gsub:cn` 58, 3113
`\int_gsub:Nn` . . 58, 3113, 3131, 3140, 10575
`\int_gzero:c` 58, 3109
`\int_gzero:N` 58, 3109, 3111, 3112
`\int_if_even:n` 3643
`\int_if_even:nF` 3772
`\int_if_even:nT` 3771
`\int_if_even:nTF` 59, 3639, 3770
`\int_if_even_p:n` 59, 3639, 3769
`\int_if_odd:n` 3639
`\int_if_odd:nF` 3767
`\int_if_odd:nT` 3766
`\int_if_odd:nTF` 59, 3639, 3765
`\int_if_odd_p:n` 59, 3639, 3764
`\int_incr:c` 57, 3079
`\int_incr:N` 57, 3079, 3079, 3094,
3101, 3105, 5691, 5731, 6383, 6623
`\int_max:nn` 56, 3578, 3585, 3757
`\int_min:nn` 56, 3578, 3595, 3758
`\int_mod:nn`
56, 3176, 3187, 3367, 3605, 3633, 3762
`\int_new:c` 56, 2368, 3044
`\int_new:N` 56, 3044, 3048,
3053, 3290–3294, 3676, 3686, 3736,
3737, 4825, 5393, 5597, 6362, 6559,
7130, 7132, 7134, 7136, 7138, 7140,
7153–7181, 7183–7187, 7190, 7191,
7193, 7194, 7196–7199, 10530, 10532
`\int_pre_eval_one_arg:Nn` 266, 3295, 3295
`\int_pre_eval_two_args:Nnn`
. 266, 3295, 3297
`\int_roman_lcuc_mapping:Nnn`
. 67, 3146, 3146, 3150–3157
`.int_set:c` 138
`\int_set:cn` 58, 3054
`.int_set:N` 138
`\int_set:Nn` 58, 3054, 3054,
3064, 3065, 3068, 5627, 5640, 5654,
5661, 5675, 5715, 5841, 6619, 7131,
7133, 7135, 7137, 7139, 7141, 7897,
7933, 10446, 10448, 10450, 10452,
10454, 10456, 10458, 10460, 10531
`\int_set_eq:cc` 57, 3067
`\int_set_eq:cN` 57, 3067
`\int_set_eq:Nc` 57, 3067
`\int_set_eq:NN` . 57, 3067, 3067, 3070–3072
`\int_show:c` 58, 3143, 3144
`\int_show:N` 58, 1410, 3143, 3143
`\int_sub:cn` 58, 3113
`\int_sub:Nn` . . . 58, 3113, 3119, 3136, 3139
`\int_symbol_math_conversion_rule:n` .
. 67, 3257, 3264, 3264
`\int_symbol_text_conversion_rule:n` .
. 67, 3261, 3264, 3277
`\int_to_Alph:n` 61, 3191, 3222
`\int_to_alph:n` 61, 3191, 3191
`\int_to_arabic:n` 61, 3145, 3145
`\int_to_binary:n` 62, 3469, 3478
`\int_to_hexadecimal:n` . . . 62, 3469, 3481
`\int_to_octal:n` 62, 3469, 3484
`\int_to_Roman:n` 62, 3158, 3162
`\int_to_roman:n` 62, 3158, 3158

\int_to_roman:w	\intexpr_compare:nTF	3747
. 66, 2125, 2129, 3032, 3039, 3160,	\intexpr_compare_p:n	3746
3164, 4619, 4622, 4626, 4703, 4718	\intexpr_compare_p:nNn	3751
\int_to_roman_lcuc:NN	\intexpr_div_round:nn	3761
.... 67, 3158, 3159, 3163, 3166, 3168	\intexpr_div_truncate:nn	3760
\int_to_symbol:n	\intexpr_do_until:nn	3777
62, 3253, 3253	\intexpr_do_until:nNnn	3782
\int_until_do:nn 61, 3647, 3650, 3651, 3775	\intexpr_do_while:nn	3776
\int_until_do:nNnn	\intexpr_do_while:nNnn	3781
..... 60, 3659, 3662, 3663, 3780	\intexpr_eval:n	3744
\int_use:c	\intexpr_eval:w	3739
58, 3141, 3142	\intexpr_eval_end:	3740
\int_use:N	\intexpr_if_even:nF	3772
1956, 1960, 2377, 2385, 2400, 2408,	\intexpr_if_even:nT	3771
2521, 2571, 2583, 2593, 2603, 3141,	\intexpr_if_even:nTF	3770
3141, 3142, 4354, 4357, 4363, 4366,	\intexpr_if_even_p:n	3769
4838, 4846, 5396, 5399, 5666, 5667,	\intexpr_if_odd:nF	3767
5677, 5682, 5685, 5695, 5706, 5707,	\intexpr_if_odd:nT	3766
5717, 5722, 5725, 5735, 6376, 6379,	\intexpr_if_odd:nTF	3765
6387, 6389, 6620, 7264, 7302, 7319,	\intexpr_if_odd_p:n	3764
7332, 7364, 7377, 7387, 7400, 7456,	\intexpr_max:nn	3757
7459, 7461, 7499, 7502, 7504, 7912,	\intexpr_min:nn	3758
7915, 7917, 7949, 7952, 7954, 7965,	\intexpr_mod:nn	3762
7990, 8022, 8025, 8027, 8051, 8054,	\intexpr_until_do:nn	3775
8056, 8108, 8114, 8233, 8239, 8281,	\intexpr_until_do:nNnn	3780
8291, 8376, 8383, 8394, 8616, 8627,	\intexpr_value:w	3738
8641, 8652, 8662, 8672, 8684, 8701,	\intexpr_while_do:nn	3774
8845, 8851, 8899, 8906, 8944, 8950,	\intexpr_while_do:nNnn	3779
8987, 8994, 9144, 9150, 9265, 9272,	\ior_alloc_read:n 5641, 5651, 5658	
9419, 9425, 9495, 9527, 9552, 9555,	\ior_close:c	120
9633, 9640, 9777, 9783, 9820, 9827,	\ior_close:N	
9841, 9863, 9879, 9890, 9900, 9910,	... 120, 5639, 5757, 5768, 7041, 7069	
9979, 9981, 9983, 10009, 10020,	\ior_gto:NN	122, 5828, 5831
10195, 10202, 10227, 10229, 10231,	\ior_if_eof:N	5823
10233, 10235, 10447, 10449, 10451,	\ior_if_eof:Nf	7059
10453, 10455, 10457, 10459, 10461	\ior_if_eof:nTF	122, 5823, 7038
\int_value:w 67, 3032, 3032, 3042, 3511,	\ior_if_eof_p:N	122, 5823
3579, 3586, 3596, 3609, 3634, 3738	\ior_new:c	119, 5617
\int_while_do:nn 61, 3647, 3647, 3648, 3774	\ior_new:N	119, 5617, 5621, 5624
\int_while_do:nNnn	\ior_open:cn	119, 5625
..... 60, 3659, 3659, 3660, 3779	\ior_open:Nn	
\int_zero:c 119, 5625, 5638, 5650, 7037, 7058	
58, 3109	\ior_open_streams:	5769, 5772
\int_zero:N 58, 3109, 3109, 3110, 6603	\ior_raw_new:c	122, 5722
\interactionmode	\ior_raw_new:N	
496	... 122, 5599, 5606, 5612, 5616, 5714	
\interlinepenalties	\ior_stream_alloc:N 5645, 5665, 5705	
517	\ior_stream_alloc_aux:	
\interlinepenalty 5665, 5711, 5730, 5738, 5740	
376	\ior_to:NN	122, 5828, 5828, 5832
\intexpr_abs:n		
3756		
\intexpr_compare:nF		
3749		
\intexpr_compare:nNnF		
3754		
\intexpr_compare:nNnT		
3753		
\intexpr_compare:nNnTF		
3752		
\intexpr_compare:nT		
3748		

\iow_alloc_write:n 5628, 5651, 5651
 \iow_char:N
 121, 5064, 5155, 5295, 5297, 5821, 5821
 \iow_close:c 120, 5745
 \iow_close:N .. 120, 5626, 5745, 5745, 5756
 \iow_log:n ... 120, 5793, 5794, 7105–7107
 \iow_log:x 16, 120, 1233,
 1233, 1256, 1744, 5793, 5793, 5974
 \iow_new:c 119, 5617
 \iow_new:N 119, 5617, 5617, 5620
 \iow_newline: 121, 5063, 5820, 5820
 \iow_now:Nn
 120, 5790, 5790, 5794, 5796, 5798, 5809
 \iow_now:Nx 120, 5789,
 5789, 5791, 5793, 5795, 5802, 5806
 \iow_now_buffer_safe:Nn 120, 5805, 5805
 \iow_now_buffer_safe:Nx 120, 5805, 5808
 \iow_now_buffer_safe_aux:w
 5806, 5809, 5811
 \iow_now_buffer_safe_expanded_aux:w
 5805
 \iow_now_when_avail:cn 121, 5797
 \iow_now_when_avail:cx 121, 5797
 \iow_now_when_avail:Nn
 121, 5797, 5797, 5800
 \iow_now_when_avail:Nx
 121, 5797, 5801, 5804
 \iow_open:cn 119, 5625
 \iow_open:Nn .. 119, 5625, 5625, 5637, 6304
 \iow_open_streams: 120, 5769, 5769
 \iow_raw_new:c 122, 5682
 \iow_raw_new:N
 122, 5599, 5602, 5611, 5615, 5674
 \iow_shipout:Nn ... 121, 5816, 5816, 5819
 \iow_shipout:Nx 121, 5816
 \iow_shipout_x:Nn
 16, 121, 881, 881, 1234, 1237,
 5789, 5814, 5814, 5815, 5817, 6292
 \iow_shipout_x:Nx 121, 5814
 \iow_stream_alloc:N 5632, 5665, 5665
 \iow_stream_alloc_aux:
 5665, 5671, 5690, 5698, 5700
 \iow_term:n 120, 5793, 5796
 \iow_term:x 16, 120,
 1233, 1236, 1240, 1255, 5046, 5050,
 5147, 5287, 5793, 5795, 5944, 5981

 J
 \jobname 451, 6304, 6308

K

\K 2710
 \kern 329
 \kernel_register_show:c 1401, 1410, 3144
 \kernel_register_show:N 1401,
 1401, 3143, 3870, 3958, 4048, 4672
 \keys_bool_set:Nn
 142, 6568, 6568, 6836, 6839
 \keys_choice_code_store:x
 142, 6584, 6584, 6845, 6848
 \keys_choice_find:n 143, 6587, 6587, 6594
 \keys_choice_make:
 142, 6578, 6592, 6592, 6602, 6842
 \keys_choices_generate:n
 142, 6601, 6601, 6887
 \keys_choices_generate_aux:n
 6601, 6614, 6616
 \keys_cmd_set:nn
 143, 6593, 6596, 6625, 6625, 6723, 6851
 \keys_cmd_set:nx ... 143, 6569, 6573,
 6617, 6625, 6630, 6727, 6830, 6854
 \keys_cmd_set_aux:n 6625, 6626, 6631, 6635
 \keys_default_set:n
 143, 6582, 6641, 6641, 6646, 6857
 \keys_default_set:V 143, 6641, 6860
 \keys_define:nn ... 136, 6647, 6647, 6966
 \keys_define_aux:nnn ... 6647, 6650, 6656
 \keys_define_aux:onnn 6647, 6648
 \keys_define_elt:n . 143, 6653, 6657, 6657
 \keys_define_elt:nn 143, 6653, 6657, 6661
 \keys_define_elt_aux:nn
 6659, 6663, 6665, 6665
 \keys_define_key:n . 143, 6670, 6676, 6676
 \keys_execute: 143, 6691, 6691, 6799, 6805
 \keys_execute_aux:nn
 6588, 6589, 6691, 6692, 6697, 6702
 \keys_execute_unknown:
 143, 6691, 6693, 6696
 \keys_if_exist:nn 6710
 \keys_if_exist:nnTF 142, 6710
 \keys_if_value_requirement:nTF
 143, 6717, 6717, 6785, 6797
 \keys_meta_make:n . 143, 6722, 6722, 6902
 \keys_meta_make:x . 143, 6722, 6726, 6905
 \keys_property_find:n 143, 6666, 6731, 6731
 \keys_property_find_aux:n 6731, 6734, 6739
 \keys_property_find_aux:w
 6731, 6740, 6742, 6747
 \keys_property_new:nn
 144, 6753, 6753, 6841, 6943, 6946

- \keys_property_new_arg:nn . 144, 6753, 6756, 6835, 6838, 6844, 6847, 6850, 6853, 6856, 6859, 6862, 6865, 6868, 6871, 6874, 6877, 6880, 6883, 6886, 6889, 6892, 6895, 6898, 6901, 6904, 6907, 6910, 6913, 6916, 6919, 6922, 6925, 6928, 6931, 6934, 6937, 6940
 - \keys_property_undefine:n 144, 6636, 6759, 6759
 - \keys_set:nn 141, 6724, 6728, 6762, 6762, 6765
 - \keys_set:nV 141, 6762
 - \keys_set:nv 141, 6762
 - \keys_set_aux:nnn 6762, 6766, 6772
 - \keys_set_aux:onn 6762, 6763
 - \keys_set_elt:n ... 144, 6769, 6773, 6773
 - \keys_set_elt:nn .. 144, 6769, 6773, 6777
 - \keys_set_elt_aux: 6781, 6790, 6793, 6796
 - \keys_set_elt_aux:nn 6775, 6779, 6781, 6781
 - \keys_show:nn 142, 6808, 6808
 - \keys_tmp:w 144, 6667, 6669, 6682, 6688, 6703–6705, 6811, 6811
 - \keys_value_or_default:n 144, 6784, 6812, 6812
 - \keys_value_requirement:n 144, 6822, 6822, 6944, 6947
 - \keys_variable_set:cnNN .. 144, 6826, 6866, 6872, 6878, 6884, 6893, 6899, 6911, 6917, 6923, 6929, 6935, 6941
 - \keys_variable_set:NnNN 144, 6826, 6826, 6834, 6863, 6869, 6875, 6881, 6890, 6896, 6908, 6914, 6920, 6926, 6932, 6938
 - \KV_add_element_aux:w .. 6430, 6435, 6435
 - \KV_add_value_element:w 385, 6404, 6413, 6510
 - \KV_key_no_value_elt:n ... 134, 6377, 6381, 6386, 6472, 6517, 6527, 6527
 - \KV_key_value_elt:nn 134, 6380, 6382, 6388, 6490, 6522, 6527, 6528
 - \KV_parse_elt:w 6442, 6450, 6454, 6454, 6456, 6461
 - \KV_parse_no_sanitization_aux:n 6447, 6447, 6531, 6539
 - \KV_parse_no_space_removal_no_sanitization:n 133, 6372, 6529, 6529
 - \KV_parse_sanitization_aux:n 6436, 6436, 6535
 - \KV_parse_space_removal_no_sanitization:n 134, 6369, 6533, 6537
 - \KV_parse_space_removal_sanitization:n 134, 6366, 6533, 6533
 - \KV_process_aux:NNNn 6365, 6366, 6369, 6372, 6374
 - \KV_process_no_space_removal_no_sanitization:NNn 133, 6365, 6371, 6652
 - \KV_process_space_removal_no_sanitization:NNn 133, 6365, 6368
 - \KV_process_space_removal_sanitization:NNn 133, 6365, 6365, 6768
 - \KV_remove_surrounding_spaces:nw ... 385, 6404, 6406
 - \KV_remove_surrounding_spaces_auxi:w ... 385, 6404, 6407, 6409, 6415, 6424
 - \KV_remove_surrounding_spaces_auxii:w 6404, 6410, 6412
 - \KV_sanitization_outerlevel_active_commas:N 385, 6391, 6400, 6441
 - \KV_sanitization_outerlevel_active_equals:N 385, 6391, 6397, 6440
 - \KV_set_key_element:w 385, 6404, 6421, 6467
 - \KV_split_key_value_current:w 385, 6460, 6465, 6465, 6530, 6534, 6538
 - \KV_split_key_value_no_space_removal:w 385, 6513, 6513, 6530
 - \KV_split_key_value_space_removal:w 385, 6466, 6466, 6534, 6538
 - \KV_split_key_value_space_removal_aux:w 6466, 6478, 6487
 - \KV_split_key_value_space_removal_detect_error:wTF 385, 6466, 6477, 6483
 - \KV_val_preserve_braces:NnN . 6486, 6506
- L**
- \L 2710
 - \l_cctab_tmp_tl 10565, 10576–10579, 10593
 - \l_clist_remove_clist 5214, 5214, 5216, 5218, 5221, 5222, 5238, 5241, 5245
 - \l_exp_tl ... 1569, 1580, 1580, 1607, 1608
 - \l_file_name_tl 408, 7030, 7030, 7073, 7074, 7080, 7081, 7091
 - \l_file_search_path_saved_seq 408, 7032, 7033, 7051, 7067
 - \l_file_search_path_seq 408, 7031, 7031, 7051, 7053, 7056, 7067, 7097, 7098, 7101
 - \l_fp_arg_tl .. 7152, 7152, 8838, 8857, 8861, 8871, 8891, 8892, 8937, 8952, 8960, 8979, 8980, 9137, 9156, 9160, 9170, 9179, 9202, 9232, 9257, 9258,

- 9412, 9427, 9436, 9438, 9465, 9503,
 9630, 9774, 9785, 9793, 9812, 9813
 \l_fp_count_int 7153, 7153, 8388, 8422,
 8433, 8440, 9033, 9064, 9080, 9082,
 9085, 9087, 9607, 9644, 9652, 9906,
 9909, 9910, 9932, 9973, 10029, 10040
 \l_fp_div_offset_int 7154,
 7154, 8345, 8399, 8440, 8442, 9246
 \l_fp_exp_decimal_int ... 7155, 7156,
 9487, 9520, 9539, 9558, 9575, 9583,
 9588, 9594, 9610, 9659, 9664, 9668,
 9671, 9675, 9679, 9682, 9684, 9850,
 9866, 9876, 9879, 9886, 9894, 9919,
 9978, 9986, 9990, 10000, 10041, 10042
 \l_fp_exp_exponent_int
 7155, 7158, 9489, 9522, 9544, 9563,
 9577, 9848, 9852, 9875, 9903, 9982
 \l_fp_exp_extended_int 7155,
 7157, 9488, 9521, 9539, 9558, 9576,
 9584, 9587, 9592, 9598, 9610, 9660,
 9662, 9665, 9676, 9678, 9680, 9851,
 9866, 9896, 9900, 9902, 9919, 9980,
 9987, 9989, 9992, 10000, 10041, 10043
 \l_fp_exp_integer_int
 7155, 7155, 9486, 9519, 9539, 9558,
 9574, 9582, 9586, 9610, 9670, 9683,
 9849, 9866, 9885, 9890, 9893, 9919
 \l_fp_input_a_decimal_int
 7159, 7161, 7210,
 7394, 7395, 7400, 7402, 7443, 7445,
 7459, 7486, 7488, 7502, 7901, 7915,
 7938, 7952, 7963, 7965, 7971, 7975,
 8015, 8025, 8043, 8054, 8126, 8141,
 8245, 8327, 8392, 8394, 8397, 8412,
 8424, 8425, 8427, 8450, 8614, 8616,
 8624, 8631, 8632, 8638, 8641, 8648,
 8656, 8731, 8744, 8745, 8749, 8752,
 8754, 8760, 8768, 8770, 8782, 8783,
 8790, 8792, 8799, 8802, 8808, 8810,
 8814, 8836, 8849, 8935, 8948, 9020,
 9026, 9027, 9056, 9059, 9062, 9072,
 9076, 9135, 9148, 9200, 9223, 9228,
 9230, 9409, 9423, 9580, 9583, 9590,
 9596, 9605, 9647, 9745, 9750, 9781,
 9925, 9938, 9942, 9945, 9958, 9963,
 9967, 9970, 9971, 10031, 10033,
 10036, 10039, 10072, 10078, 10086,
 10103, 10128, 10160, 10210, 10220,
 10240, 10244, 10275, 10292, 10308,
 10332, 10400, 10431, 10450, 10459
 \l_fp_input_a_exponent_int
 ... 7159, 7162, 7211, 7353, 7360,
 7383, 7403, 7444, 7461, 7487, 7504,
 7917, 7934, 7954, 7976, 8027, 8056,
 8092, 8206, 8349, 8612, 8634, 8637,
 8665, 8707, 8837, 8851, 8853, 8936,
 8950, 9136, 9150, 9152, 9176, 9224,
 9229, 9250, 9410, 9425, 9447, 9746,
 9783, 9831, 9832, 9837, 9841, 9843,
 9845, 9862, 9863, 9956, 9964, 10073,
 10079, 10123, 10156, 10211, 10221,
 10246, 10276, 10293, 10310, 10382,
 10386, 10413, 10417, 10452, 10461
 \l_fp_input_a_extended_int
 7167, 7167, 8625, 8627, 8633, 8658,
 8662, 8664, 8708, 8746-8748, 8750,
 8760, 8772, 8774, 8784, 8791, 8793,
 8800, 8803, 8811, 8815, 9026, 9027,
 9060, 9063, 9072, 9076, 9108, 9411,
 9584, 9600, 9606, 9647, 9677, 9907,
 9939, 9946, 9965, 9968, 9970, 9971,
 10031, 10033, 10036, 10039, 10122,
 10128, 10158, 10238, 10241, 10245
 \l_fp_input_a_integer_int
 7159, 7160, 7209, 7384, 7387, 7393,
 7442, 7456, 7485, 7499, 7912, 7949,
 7970, 7972, 7974, 8022, 8051, 8122,
 8137, 8255, 8260, 8264, 8269, 8327,
 8391, 8397, 8406, 8409, 8423, 8426,
 8446, 8448, 8613, 8622, 8623, 8647,
 8652, 8655, 8711, 8713, 8725, 8730,
 8742, 8743, 8753, 8756, 8762, 8764,
 8766, 8790, 8792, 8797, 8799, 8802,
 8810, 8814, 8835, 8845, 8934, 8944,
 9134, 9144, 9201, 9222, 9227, 9231,
 9408, 9419, 9450, 9460, 9474, 9485,
 9495, 9497, 9499, 9523, 9527, 9529,
 9531, 9550, 9552, 9555, 9744, 9750,
 9777, 9925, 9931, 9941, 9944, 9955,
 9962, 10071, 10077, 10086, 10103,
 10161, 10209, 10219, 10240, 10244,
 10274, 10291, 10307, 10332, 10390,
 10395, 10421, 10426, 10448, 10457
 \l_fp_input_a_sign_int 7159,
 7159, 7205, 7207, 7441, 7451, 7484,
 7494, 7907, 7944, 8046, 8085, 8119,
 8162, 8220, 8363, 8712, 8717, 8757,
 8834, 8840, 8887, 8894, 8933, 8939,
 8975, 8982, 9008, 9010, 9015, 9133,
 9139, 9186, 9226, 9407, 9414, 9449,

- 9501, 9533, 9551, 9581, 9604, 9645,
9743, 9747, 10070, 10076, 10153,
10207, 10273, 10290, 10306, 10352,
10365, 10369, 10373, 10446, 10455
- \l_fp_input_b_decimal_int
... [7159](#), 7165, 7371, 7372, 7377,
7379, 8076, 8126, 8141, 8180, 8201,
8247, 8313, 8317, 8412, 8424, 9214,
9219, 9605, 9648, 9649, 9651, 9653,
9656, 9659, 9675, 9958, 9972, 10032,
10072, 10082, 10213, 10220, 10230,
10242, 10281, 10292, 10308, 10335,
10350, 10400, 10431, 10451, 10458
- \l_fp_input_b_exponent_int
..... [7159](#), 7166, 7353,
7360, 7380, 7383, 8077, 8181, 8202,
8206, 8314, 8349, 9215, 9220, 9250,
9959, 10073, 10214, 10221, 10234,
10246, 10282, 10293, 10310, 10382,
10386, 10413, 10417, 10453, 10460
- \l_fp_input_b_extended_int ... [7167](#),
7168, 9606, 9648, 9649, 9651, 9653,
9656, 9661, 9972, 10032, 10232, 10243
- \l_fp_input_b_integer_int [7159](#),
7164, 7361, 7364, 7370, 8075, 8122,
8137, 8179, 8200, 8258, 8262, 8265,
8269, 8312, 8317, 8406, 8409, 8423,
9213, 9218, 9957, 10071, 10082,
10212, 10219, 10228, 10242, 10280,
10291, 10307, 10335, 10350, 10390,
10395, 10421, 10426, 10449, 10456
- \l_fp_input_b_sign_int
..... [7159](#), 7163, 8074, 8085,
8148, 8178, 8182, 8199, 8220, 8311,
8363, 9217, 9604, 9645, 9658, 10070,
10118, 10226, 10279, 10290, 10306,
10340, 10365, 10369, 10447, 10454
- \l_fp_mul_a_i_int
..... [7169](#), 7169, 8246, 8251,
8256, 8261, 8265, 8491, 8500, 8507,
8513, 8518, 8524, 8527, 8534, 8543,
8551, 8559, 8566, 8574, 8579, 8583
- \l_fp_mul_a_ii_int [7169](#),
7170, 8246, 8252, 8257, 8262, 8491,
8501, 8508, 8514, 8519, 8525, 8534,
8544, 8552, 8560, 8567, 8575, 8580
- \l_fp_mul_a_iii_int
..... [7169](#), 7171, 8246, 8253,
8258, 8491, 8502, 8509, 8515, 8520,
8534, 8545, 8553, 8561, 8568, 8576
- \l_fp_mul_a_iv_int
... [7169](#), 7172, 8493, 8503, 8510,
8516, 8536, 8546, 8554, 8562, 8569
- \l_fp_mul_a_v_int .. [7169](#), 7173, 8493,
8504, 8511, 8536, 8547, 8555, 8563
- \l_fp_mul_a_vi_int [7169](#),
7174, 8493, 8505, 8536, 8548, 8556
- \l_fp_mul_b_i_int
..... [7169](#), 7175, 8248, 8253,
8257, 8261, 8264, 8495, 8505, 8511,
8516, 8520, 8525, 8527, 8538, 8548,
8555, 8562, 8568, 8575, 8579, 8582
- \l_fp_mul_b_ii_int [7169](#),
7176, 8248, 8252, 8256, 8260, 8495,
8504, 8510, 8515, 8519, 8524, 8538,
8547, 8554, 8561, 8567, 8574, 8578
- \l_fp_mul_b_iii_int
..... [7169](#), 7177, 8248, 8251,
8255, 8495, 8503, 8509, 8514, 8518,
8538, 8546, 8553, 8560, 8566, 8573
- \l_fp_mul_b_iv_int
... [7169](#), 7178, 8497, 8502, 8508,
8513, 8540, 8545, 8552, 8559, 8565
- \l_fp_mul_b_v_int .. [7169](#), 7179, 8497,
8501, 8507, 8540, 8544, 8551, 8558
- \l_fp_mul_b_vi_int [7169](#),
7180, 8497, 8500, 8540, 8543, 8550
- \l_fp_mul_output_int [7181](#),
7181, 8249, 8254, 8285, 8286, 8289,
8291, 8296, 8498, 8506, 8541, 8549
- \l_fp_mul_output_tl
... [7181](#), 7182, 8250, 8267, 8268,
8271, 8295, 8499, 8522, 8523, 8530,
8542, 8571, 8572, 8585, 8586, 8589
- \l_fp_output_decimal_int
..... [7183](#), 7185, 8096, 8112,
8124, 8128, 8131, 8139, 8143, 8145,
8149, 8152, 8154, 8211, 8224, 8237,
8267, 8343, 8354, 8367, 8380, 8437,
8439, 8679, 8684, 8691, 8719, 8882,
8883, 8889, 8903, 8970, 8971, 8977,
8991, 9022, 9037, 9041, 9046, 9050,
9057, 9059, 9090, 9095, 9099, 9102,
9106, 9110, 9113, 9115, 9213, 9222,
9244, 9255, 9269, 9478, 9520, 9540,
9542, 9559, 9561, 9611, 9613, 9618,
9619, 9621, 9628, 9637, 9800, 9801,
9803, 9810, 9824, 9850, 9855, 9867,
9869, 9917, 9920, 9963, 9966, 9967,
9979, 9998, 10001, 10006, 10009,

- 10016, 10024, 10042, 10046, 10050,
10053, 10199, 10213, 10231, 10248
- \l_fp_output_exponent_int [7183](#), [7186](#),
8092, 8097, 8114, 8204, 8212, 8239,
8347, 8355, 8383, 8704, 8720, 8880,
8884, 8890, 8906, 8968, 8972, 8978,
8994, 9248, 9256, 9272, 9480, 9522,
9544, 9563, 9629, 9640, 9811, 9827,
9852, 9857, 9875, 9964, 9983, 10005,
10026, 10202, 10214, 10235, 10249
- \l_fp_output_extended_int
..... [7187](#), [7187](#), 8695, 8696,
8701, 8703, 8883, 8971, 9038, 9042,
9047, 9049, 9060, 9091, 9093, 9096,
9107, 9109, 9111, 9214, 9223, 9479,
9521, 9541, 9543, 9560, 9562, 9612,
9614, 9616, 9798, 9851, 9856, 9868,
9870, 9918, 9921, 9968, 9981, 9999,
10002, 10043, 10044, 10047, 10233
- \l_fp_output_integer_int
[7183](#), [7184](#), 8095, 8108, 8120, 8130,
8135, 8144, 8147, 8150, 8156, 8158,
8210, 8224, 8233, 8271, 8342, 8353,
8367, 8376, 8433, 8668, 8669, 8672,
8678, 8718, 8879, 8882, 8888, 8899,
8967, 8970, 8976, 8987, 9021, 9036,
9040, 9045, 9055, 9101, 9114, 9243,
9254, 9265, 9477, 9502, 9519, 9540,
9542, 9559, 9561, 9611, 9613, 9622,
9627, 9633, 9804, 9809, 9820, 9849,
9854, 9867, 9869, 9917, 9920, 9962,
9998, 10001, 10015, 10020, 10023,
10052, 10195, 10212, 10229, 10247
- \l_fp_output_sign_int
... [7183](#), [7183](#), 8094, 8103, 8119,
8148, 8162, 8209, 8352, 9188, 9190,
9194, 9196, 9253, 9260, 9626, 9808,
9815, 9833, 9835, 9912, 9993, 10227
- \l_fp_round_carry_bool [7188](#),
[7188](#), 7960, 7969, 7981, 7987, 7995
- \l_fp_round_decimal_tl [7189](#),
[7189](#), 7962, 7971, 7989, 7990, 7992
- \l_fp_round_position_int [7190](#),
[7190](#), 7961, 7980, 7993, 7999, 8000
- \l_fp_round_target_int .. [7190](#), [7191](#),
7897, 7898, 7933, 7935, 7980, 7993
- \l_fp_sign_tl
.... [7192](#), [7192](#), 10119, 10131, 10194
- \l_fp_split_sign_int
..... [7193](#), [7193](#), 7217, 7219, 7231
- \l_fp_tmp_dim [7468](#), [7477](#), [7481](#), [7516](#)
- \l_fp_tmp_int
... [7194](#), [7194](#), 7262, 7264, 7982–
7985, 7990, 8592–8594, 8599–8601
- \l_fp_tmp_skip ... [7468](#), [7476](#), [7477](#), [7517](#)
- \l_fp_tmp_tl .. [7195](#), [7195](#), 7214–7216,
[7220](#), [7224](#), [7226](#), [7229](#), [7235](#), [7237](#),
[7240](#), [7329](#), [7334](#), [7373](#), [7379](#), [7396](#),
[7402](#), [8044](#), [8059](#), [8650](#), [8656](#), [8659](#),
[8664](#), [8680](#), [8687](#), [8697](#), [8703](#), [9492](#),
[9499](#), [9506](#), [9512](#), [9524](#), [9531](#), [9534](#),
[9536](#), [9838](#), [9845](#), [9847](#), [9888](#), [9894](#),
[9897](#), [9902](#), [10018](#), [10024](#), [10444](#), [10463](#)
- \l_fp_trig_decimal_int [7197](#),
[7198](#), 9028, 9030, 9032, 9035, 9050,
9062, 9071, 9073, 9075, 9077, 9079,
9081, 9084, 9086, 9088, 9090, 9106
- \l_fp_trig_extended_int
... [7197](#), [7199](#), 9028, 9030, 9032,
9034, 9049, 9063, 9071, 9073, 9075,
9077, 9079, 9081, 9084, 9086, 9092
- \l_fp_trig_octant_int
[7196](#), [7196](#), 8780, 8786, 8804, 8816,
8910, 8998, 9009, 9013, 9187, 9193
- \l_fp_trig_sign_int
.. [7197](#), [7197](#), 9024, 9061, 9069, 9089
- \l_int_from_roman_l_int [3722](#)
- \l_ior_stream_int
.... [123](#), [5597](#), 5598, 5640, 5642,
5646, 5661, 5706, 5707, 5712, 5715,
5722, 5725, 5731, 5733, 5735, 5737
- \l_iow_stream_int [123](#), [5597](#),
5597, 5598, 5627, 5629, 5633, 5654,
5666, 5667, 5672, 5675, 5677, 5682,
5685, 5691, 5693, 5695, 5697, 5717
- \l_kernel_testa_tl .. [83](#), [4249](#), 4249, 5101
- \l_kernel_testb_tl [83](#), [4249](#), 4250
- \l_kernel_tmpa_tl [83](#), 696, 698,
699, 710, 712, [4253](#), 4253, 4439, 4442
- \l_kernel_tmpb_tl [83](#), 697, 698, [4253](#), 4254
- \l_keys_choice_code_tl
..... [145](#), [6561](#), 6561, 6607, 6621
- \l_keys_choice_int
[141](#), [6559](#), 6559, 6603, 6619, 6620, 6623
- \l_keys_choice_tl . [141](#), [6559](#), 6560, 6618
- \l_keys_key_tl [142](#), [6562](#), 6562, 6782, 6783
- \l_keys_module_tl .. [145](#), [6565](#), 6565,
6648, 6651, 6654, 6697, 6699, 6724,
6728, 6732, 6763, 6767, 6770, 6783

- \l_keys_no_value_bool
 [145](#), [6566](#), [6566](#), [6658](#), [6662](#),
 [6677](#), [6774](#), [6778](#), [6786](#), [6798](#), [6814](#)
- \l_keys_path_tl [145](#),
 [6562](#), [6563](#), [6569](#), [6573](#), [6585](#), [6588](#),
 [6589](#), [6593](#), [6596](#), [6598](#), [6605](#), [6608](#),
 [6612](#), [6617](#), [6642](#)–[6644](#), [6673](#), [6685](#),
 [6692](#), [6698](#), [6719](#), [6723](#), [6727](#), [6732](#),
 [6744](#), [6745](#), [6749](#), [6783](#), [6788](#), [6802](#),
 [6815](#), [6817](#), [6823](#), [6830](#), [6851](#), [6854](#)
- \l_keys_property_tl [145](#), [6562](#),
 [6564](#), [6668](#), [6673](#), [6680](#), [6685](#), [6750](#)
- \l_keys_value_tl
 [145](#), [6567](#), [6567](#), [6705](#), [6802](#), [6813](#), [6816](#)
- \l_KV_currkey_tl [134](#), [6359](#),
 [6360](#), [6422](#), [6429](#), [6430](#), [6473](#), [6491](#)
- \l_KV_currval_tl [134](#), [6359](#), [6361](#), [6414](#), [6418](#)
- \l_KV_level_int [6362](#), [6362](#),
 [6376](#), [6379](#), [6383](#), [6385](#), [6387](#), [6389](#)
- \l_KV_parse_tl [134](#), [6359](#), [6359](#),
 [6417](#), [6438](#), [6445](#), [6449](#), [6452](#), [6471](#),
 [6489](#), [6494](#), [6500](#), [6505](#), [6517](#), [6522](#)
- \l_KV_remove_one_level_of_braces_bool
 ... [133](#), [6363](#), [6363](#), [6364](#), [6427](#), [6498](#)
- \l_KV_tmpa_tl [134](#), [6356](#),
 [6356](#), [6439](#)–[6441](#), [6443](#), [6468](#), [6469](#),
 [6488](#), [6493](#), [6496](#), [6514](#), [6515](#), [6520](#)
- \l_KV_tmpb_tl [134](#), [6356](#), [6357](#)
- \l_last_box [115](#), [5454](#), [5454](#), [5455](#)
- \l_msg_class_tl
 ... [130](#), [5886](#), [5886](#), [6065](#), [6066](#), [6069](#)
- \l_msg_coding_error_text_tl
 ... [6958](#), [6980](#), [6987](#)
- \l_msg_current_class_tl
 ... [130](#), [5886](#), [5887](#), [6045](#), [6066](#)
- \l_msg_current_module_tl
 ... [130](#), [5886](#), [5888](#), [6046](#)
- \l_msg_names_clist
 ... [130](#), [5889](#), [5889](#), [5923](#), [5924](#)
- \l_msg_redirect_classes_clist
 ... [130](#), [5892](#), [5892](#), [6021](#), [6025](#), [6028](#)
- \l_msg_redirect_classes_prop
 ... [130](#), [5890](#), [5890](#)
- \l_msg_redirect_names_prop
 ... [130](#), [5890](#), [5891](#), [6047](#), [6134](#)
- \l_msg_tmp_tl [5893](#), [5893](#), [5942](#), [5961](#), [5963](#)
- \l_peek_false_tl ... [247](#), [2873](#), [2874](#),
 [2881](#), [2896](#), [2914](#), [2921](#), [2930](#), [2937](#)
- \l_peek_search_tl
 ... [247](#), [2876](#), [2876](#), [2879](#), [2893](#), [2931](#)
- \l_peek_search_token
 ... [53](#), [2866](#), [2868](#), [2878](#), [2892](#), [2911](#), [2918](#)
- \l_peek_token [53](#), [2866](#), [2866](#),
 [2869](#), [2911](#), [2918](#), [2926](#)–[2928](#), [3019](#)
- \l_peek_true_aux_tl
 ... [247](#), [2894](#), [2906](#), [2906](#), [2908](#)
- \l_peek_true_tl [247](#), [2873](#),
 [2873](#), [2880](#), [2895](#), [2912](#), [2919](#), [2935](#)
- \l_seq_remove_seq
 ... [4998](#), [4998](#), [5005](#), [5008](#), [5009](#), [5011](#)
- \l_seq_show_tl ... [5041](#), [5041](#), [5055](#), [5058](#)
- \l_seq_tmpa_tl
 ... [4758](#), [4758](#), [4979](#), [4983](#), [5030](#), [5035](#)
- \l_seq_tmpb_tl
 ... [4758](#), [4759](#), [4982](#), [4983](#), [5026](#), [5030](#)
- \l_tl_replace_tl [83](#)
- \l_tl_replace_toks [90](#), [4745](#), [4745](#)
- \l_tl_tmpa_tl [4270](#), [4272](#), [4274](#), [4282](#)
- \l_tl_tmpb_tl [4270](#), [4273](#), [4274](#), [4283](#)
- \l_tmpa_bool [222](#), [2020](#), [2020](#)
- \l_tmpa_box [115](#), [5491](#), [5492](#), [5494](#)
- \l_tmpa_dim [75](#), [3960](#), [3960](#)
- \l_tmpa_int [65](#), [3290](#), [3290](#)
- \l_tmpa_skip [71](#), [3875](#), [3875](#)–[3877](#)
- \l_tmpa_tl [5](#), [83](#), [4251](#), [4251](#), [4426](#), [4432](#),
 [4459](#), [5202](#), [5205](#), [5206](#), [5939](#), [5949](#)
- \l_tmpa_toks [90](#), [4737](#), [4737](#),
 [5149](#), [5151](#), [5152](#), [5154](#), [5158](#), [5289](#),
 [5291](#), [5292](#), [5294](#), [5300](#), [6257](#), [6263](#)
- \l_tmpb_box [115](#), [5491](#), [5495](#)
- \l_tmpb_dim [75](#), [3960](#), [3961](#)
- \l_tmpb_int [65](#), [3290](#), [3291](#)
- \l_tmpb_skip [71](#), [3875](#), [3878](#)
- \l_tmpb_tl [83](#), [4251](#),
 [4252](#), [5203](#), [5206](#), [5207](#), [5940](#), [5949](#)
- \l_tmpb_toks [90](#), [4737](#), [4739](#)
- \l_tmpc_dim [75](#), [3960](#), [3962](#)
- \l_tmpc_int [65](#), [3290](#), [3292](#)
- \l_tmpc_skip [71](#), [3875](#), [3879](#)
- \l_tmpc_toks [90](#), [4737](#), [4740](#)
- \l_tmpd_dim [75](#), [3960](#), [3963](#)
- \l_xref_curr_name_tl [6314](#)
- \language [246](#)
- \lastbox [403](#)
- \lastkern [336](#)
- \lastlinefit [516](#)
- \lastnodetype [497](#)
- \lastpenalty [442](#)
- \lastskip [337](#)
- \latelua [639](#)

<code>\lccode</code>	465	<code>\mark</code>	247
<code>\leaders</code>	333	<code>\marks</code>	473
<code>\left</code>	301	<code>\mathaccent</code>	258
<code>\lefthyphenmin</code>	357	<code>\mathbin</code>	288
<code>\leftmargin</code>	579	<code>\mathchar</code>	259
<code>\leftskip</code>	359	<code>\mathchardef</code>	156
<code>\leqno</code>	276	<code>\mathchoice</code>	256
<code>\let</code>	49, 59, 122, 146	<code>\mathclose</code>	289
<code>\limits</code>	293	<code>\mathcode</code>	467
<code>\linepenalty</code>	349	<code>\mathinner</code>	290
<code>\lineskip</code>	343	<code>\mathop</code>	291
<code>\lineskiplimit</code>	344	<code>\mathopen</code>	295
<code>\long</code>	32, 125, 131, 134, 139, 165	<code>\mathord</code>	296
<code>\looseness</code>	361	<code>\mathparagraph</code>	3270
<code>\lower</code>	398	<code>\mathpunct</code>	297
<code>\lowercase</code>	437	<code>\mathrel</code>	298
<code>\lpcode</code>	546	<code>\mathsection</code>	3269
<code>\lua_now:n</code>	156, 10506, 10515	<code>\mathsurround</code>	309
<code>\lua_now:x</code>	156, 10506, 10508, 10512, 10516	<code>\maxdeadcycles</code>	379
<code>\lua_shipout:n</code>	157, 10506, 10519, 10522	<code>\maxdepth</code>	380
<code>\lua_shipout:x</code>	157, 10506	<code>\maxdimen</code>	3890
<code>\lua_shipout_x:n</code>	157, 10506, 10509, 10513, 10518, 10520	<code>\meaning</code>	439
<code>\lua_shipout_x:x</code>	157, 10506	<code>\medmuskip</code>	310
<code>\lua_wrong_engine:</code>	10506, 10512, 10513, 10526, 10554, 10582, 10583, 10601	<code>\message</code>	216
<code>\luaescapestring</code>	37, 38	<code>\MessageBreak</code>	67–73, 833
<code>\LuaTeX</code>	10527	<code>.meta:n</code>	139
<code>\luatex_catcodetable:D</code>	636, 792, 10566, 10567, 10572, 10579	<code>.meta:x</code>	139
<code>\luatex_directlua:D</code>	637, 1418, 10508	<code>\middle</code>	521
<code>\luatex_if_engine:</code>	1425	<code>\mkern</code>	263
<code>\luatex_if_engine:F</code>	10553, 10581, 10600	<code>\mode_if_horizontal:</code>	1866
<code>\luatex_if_engine:T</code>	10557, 10586, 10603, 10608, 10616	<code>\mode_if_horizontal:TF</code>	39, 1866
<code>\luatex_if_engine:TF</code>	25, 1411, 10506	<code>\mode_if_horizontal_p:</code>	39, 1866
<code>\luatex_initcatcodetable:D</code>	638, 793, 10542, 10561	<code>\mode_if_inner:</code>	1869
<code>\luatex_latelua:D</code>	639, 794, 10509	<code>\mode_if_inner:TF</code>	39, 1869
<code>\luatex_savecatcodetable:D</code>	640, 795, 10571, 10597	<code>\mode_if_inner_p:</code>	39, 1869
<code>\luatexcatcodetable</code>	792	<code>\mode_if_math:</code>	1872
<code>\luatexinitcatcodetable</code>	793	<code>\mode_if_math:TF</code>	39, 1872, 3254
<code>\luatexlatelua</code>	794	<code>\mode_if_math_p:</code>	39, 1872
<code>\luatexsavecatcodetable</code>	795	<code>\mode_if_vertical:</code>	1863
		<code>\mode_if_vertical:TF</code>	39, 1863
		<code>\mode_if_vertical_p:</code>	39, 1863
		<code>\month</code>	449
		<code>\moveleft</code>	399
		<code>\moveright</code>	400
		<code>\msg_class_new:nn</code>	124, 5996, 5996, 6072, 6083, 6100, 6107, 6114, 6120, 6126
		<code>\msg_class_set:nn</code>	124, 5996, 5999, 6001
		<code>\msg_direct_interrupt:n</code>	5927
		<code>\msg_direct_interrupt:xxxxx</code>	128, 5927, 5935, 6073, 6084, 6169, 6182, 6233
M			
<code>\M</code>	2710, 2797		
<code>\m@ne</code>	1221		
<code>\mag</code>	245		

\msg_direct_interrupt_aux:n . 5941, 5958
 \msg_direct_log:xx
 ... 128, 5970, 5970, 6108, 6115, 6121
 \msg_direct_term:xx 128, 5970, 5977, 6101
 \msg_error:nn 125, 6083
 \msg_error:nnx 125, 6083
 \msg_error:nnxx 125, 6083
 \msg_error:nnxxx 125, 6083
 \msg_error:nnxxxx 125, 6083
 \msg_fatal:nn 124, 6072
 \msg_fatal:nnx 124, 6072
 \msg_fatal:nnxx 124, 6072
 \msg_fatal:nnxxx 124, 6072
 \msg_fatal:nnxxxx 124, 6072
 \msg_fatal_text:n
 ... 129, 5845, 5845, 6074, 6170
 \msg_generic_new:nn 127, 5904, 5908, 5988
 \msg_generic_new:nnn 127, 5904, 5904, 5985
 \msg_generic_set:nn
 ... 127, 5910, 5912, 5917, 5994
 \msg_generic_set:nnn
 ... 127, 5906, 5912, 5912, 5991
 \msg_generic_set_clist:n
 ... 5912, 5913, 5918, 5922
 \msg_info:nn 125, 6107
 \msg_info:nnx 125, 6107
 \msg_info:nnxx 125, 6107
 \msg_info:nnxxx 125, 6107
 \msg_info:nnxxxx 125, 6107, 6206
 \msg_kernel_bug:x 16, 129, 1239,
 1239, 1249, 1268, 1280, 1407, 1482,
 4077, 4808, 5102, 5121, 6232, 6232
 \msg_kernel_classes_new:n
 .. 6148, 6148, 6180, 6200, 6204, 6208
 \msg_kernel_error:nn 129,
 5630, 5643, 6026, 6181, 6479, 6524,
 10473, 10482, 10488, 10497, 10570
 \msg_kernel_error:nnx 129, 2338, 6032,
 6181, 6611, 6736, 6787, 10545, 10549
 \msg_kernel_error:nnxx ... 129, 6039,
 6181, 6597, 6672, 6684, 6698, 6801
 \msg_kernel_error:nnxxx 129, 6181
 \msg_kernel_error:nnxxxx 129, 6181, 6181
 \msg_kernel_error:nx 7418
 \msg_kernel_fatal:nn 128, 6168
 \msg_kernel_fatal:nnx 128, 6168
 \msg_kernel_fatal:nnxx 128, 6168
 \msg_kernel_fatal:nnxxx 128, 6168
 \msg_kernel_fatal:nnxxxx 128, 6168, 6168
 \msg_kernel_info:nn 129, 6201
 \msg_kernel_info:nnx 129, 6201
 \msg_kernel_info:nnxx 129, 6201
 \msg_kernel_info:nnxxx 129, 6201
 \msg_kernel_info:nnxxxx 129, 6201, 6205
 \msg_kernel_new:nnn 128, 6136, 6139
 \msg_kernel_new:nnnn 128, 5775, 5782,
 6136, 6136, 6209, 6217, 6225, 6541,
 6949, 6955, 6962, 6971, 6977, 6984,
 6991, 6997, 10466, 10475, 10484, 10490
 \msg_kernel_set:nnn 128, 6136, 6145
 \msg_kernel_set:nnnn ... 128, 6136, 6142
 \msg_kernel_warning:nn 129, 6201
 \msg_kernel_warning:nnx 129, 6201
 \msg_kernel_warning:nnxx 129, 6201
 \msg_kernel_warning:nnxxx 129, 6201
 \msg_kernel_warning:nnxxxx
 ... 129, 6201, 6201
 \msg_line_context: . 127, 5894, 5897, 6542
 \msg_line_number: . 127, 5894, 5894, 5900
 \msg_log:nn 125, 6114
 \msg_log:nnx 125, 6114
 \msg_log:nnxx 125, 6114
 \msg_log:nnxxx 125, 6114
 \msg_log:nnxxxx 125, 6114
 \msg_new:nnn 124, 5984, 5987, 6140
 \msg_new:nnnn 124, 5984, 5984, 6137
 \msg_newline:
 ... 127, 5902, 5902, 5937, 5943,
 5945, 5959, 5972, 5974, 5979, 5981
 \msg_none:nn 126, 6126
 \msg_none:nnx 126, 6126
 \msg_none:nnxx 126, 6126
 \msg_none:nnxxx 126, 6126
 \msg_none:nnxxxx 126, 6126
 \msg_redirect_class:nn . 126, 6127, 6127
 \msg_redirect_module:nnn 126, 6130, 6130
 \msg_redirect_name:nnn . 126, 6133, 6133
 \msg_see_documentation_text:n 5845,
 5846, 6079, 6090, 6176, 6189, 6237
 \msg_set:nnn 124, 5984, 5993, 6146
 \msg_set:nnnn 124, 5984, 5990, 6143
 \msg_trace:nn 126, 6120
 \msg_trace:nnx 126, 6120
 \msg_trace:nnxx 126, 6120
 \msg_trace:nnxxx 126, 6120
 \msg_trace:nnxxxx 126, 6120
 \msg_two_newlines: . 127, 5850, 5902, 5903
 \msg_use:nnnnxxxx 6004, 6019, 6019
 \msg_use_aux:nn 6050, 6053, 6053
 \msg_use_aux:nnn 6037, 6044, 6044

<code>\msg_use_code:</code>	6020, 6042 , 6042, 6060, 6067	<code>\nonscript</code>	274
<code>\msg_use_loop:</code>	6042	<code>\nonstopmode</code>	237
<code>\msg_use_loop:n</code>	6024, 6043, 6069	<code>\nulldelimiterspace</code>	307
<code>\msg_use_loop_check:nn</code>	6048 , 6055, 6058, 6064 , 6064	<code>\nullfont</code>	425
<code>\msg_warning:nn</code>	125, 6100	<code>\number</code>	434, 6320
<code>\msg_warning:nxx</code>	125, 6100	<code>\numexpr</code>	506
<code>\msg_warning:nxxx</code>	125, 6100	O	
<code>\msg_warning:nxxxx</code>	125, 6100	<code>\O</code>	1761, 2710, 2801
<code>\msg_warning:nxxxxx</code>	125, 6100 , 6202	<code>\omit</code>	180
<code>\mskip</code>	260	<code>\openin</code>	206
<code>\muexpr</code>	509	<code>\openout</code>	207
<code>\multiply</code>	161	<code>\or</code>	203
<code>\muskip</code>	457	<code>\or:</code>	8, 67, 860 , 862, 968, 1452, 1454, 1456, 1458, 1460, 1462, 1464, 1466, 1468, 3266–3274, 3279–3287
<code>\muskip_add:Nn</code>	75, 4041 , 4043, 4044	<code>\outer</code>	166
<code>\muskip_gadd:Nn</code>	75, 4041 , 4044	<code>\output</code>	381
<code>\muskip_gset:Nn</code>	75, 4041 , 4042	<code>\outputpenalty</code>	391
<code>\muskip_gsub:Nn</code>	75, 4041 , 4046	<code>\over</code>	268
<code>\muskip_new:N</code>	75, 4032 , 4036	<code>\overfullrule</code>	419
<code>\muskip_set:Nn</code>	75, 4041 , 4041, 4042	<code>\overline</code>	299
<code>\muskip_show:N</code>	76, 4048 , 4048	<code>\overwithdelims</code>	269
<code>\muskip_sub:Nn</code>	75, 4041 , 4045, 4046	P	
<code>\muskip_use:N</code>	75, 4047 , 4047	<code>\P</code>	1759, 2710
<code>\muskipdef</code>	155	<code>\p</code>	2323
<code>\mutoglu</code>	515	<code>\package_check_loaded_expl:</code>	830 , 856, 1572, 1860, 2422, 2514, 3029, 3787, 4053, 4597, 4751, 5076, 5266, 5408, 5560, 5838, 6244, 6353, 6552, 7008, 7120, 10503
N		<code>\package_provides:w</code>	803, 804
<code>\n</code>	2706	<code>\PackageError</code>	64, 832
<code>\name_pop_stack:w</code>	798, 809, 828	<code>\pagedepth</code>	383
<code>\name_primitive:NN</code>	139, 143–533, 535–555, 557–563, 565– 568, 570–591, 593–604, 606–640, 646	<code>\pagediscards</code>	524
<code>\name_tmp:</code>	828	<code>\pagefilllstretch</code>	387
<code>\name_undefine:N</code>	125, 131, 134, 141, 797	<code>\pagefillstretch</code>	386
<code>\NeedsTeXFormat</code>	820	<code>\pagefilstretch</code>	385
<code>\newbox</code>	5418	<code>\pagegoal</code>	389
<code>\newcatcodetable</code>	10560	<code>\pagenumbering</code>	6327, 6335
<code>\newcount</code>	3050	<code>\pageshrink</code>	388
<code>\newdimen</code>	3916	<code>\pagestretch</code>	384
<code>\newlinechar</code>	78, 211	<code>\pagetotal</code>	390
<code>\newmuskip</code>	4038	<code>\par</code>	339, 796
<code>\newread</code>	5612	<code>\parfillskip</code>	370
<code>\newskip</code>	3796	<code>\parindent</code>	363
<code>\newtoks</code>	4606	<code>\parshape</code>	355
<code>\newwrite</code>	5611	<code>\parshapedimen</code>	505
<code>\noalign</code>	179	<code>\parshapeindent</code>	503
<code>\noboundary</code>	314		
<code>\noexpand</code>	34, 37, 38, 110, 172		
<code>\noindent</code>	340		
<code>\nolimits</code>	294		

\parshaplength	504	\pdf_mapfile:D	624
\parskip	362	\pdf_mapline:D	625
\patterns	445	\pdf_mdffivesum:D	588
\pausing	232	\pdf_minorversion:D	536
\pdf@strcmp	49	\pdf_names:D	623
\pdf_adjustspacing:D	543	\pdf_noligatures:D	634
\pdf_annot:D	612	\pdf_normaldeviate:D	587
\pdf_catalog:D	622	\pdf_obj:D	606
\pdf_compresslevel:D	537	\pdf_optionalwaysusepdfpagebox:D	549
\pdf_creationdate:D	572	\pdf_optionpdfinclusionerrorlevel:D	551
\pdf_decimaldigits:D	538	\pdf_outline:D	615
\pdf_dest:D	616	\pdf_output:D	535
\pdf_destmargin:D	562	\pdf_pageattr:D	566
\pdf_efcode:D	545	\pdf_pageheight:D	560
\pdf_elapsedtime:D	602	\pdf_pageref:D	573
\pdf_endlink:D	614	\pdf_pageresources:D	567
\pdf_endthread:D	619	\pdf_pagesattr:D	565
\pdf_escapehex:D	583	\pdf_pagewidth:D	559
\pdf_escapename:D	582	\pdf_pkmode:D	568
\pdf_escapestring:D	581	\pdf_pkresolution:D	540
\pdf_filedump:D	591	\pdf_protrudechars:D	544
\pdf_filemoddate:D	589	\pdf_randomseed:D	603
\pdf_filesize:D	590	\pdf_refobj:D	607
\pdf_fontattr:D	626	\pdf_refxform:D	609
\pdf_fontexpand:D	628	\pdf_refximage:D	611
\pdf_fontname:D	575	\pdf_resettimer:D	632
\pdf_fontobjnum:D	576	\pdf_rightmarginkern:D	580
\pdf_fontsize:D	577	\pdf_rpcode:D	547
\pdf_forcepagebox:D	548	\pdf_savepos:D	620
\pdf_gamma:D	554	\pdf_setrandomseed:D	633
\pdf_horigin:D	557	\pdf_shellescape:D	604
\pdf_imageapplygamma:D	553	\pdf_startlink:D	613
\pdf_imagegamma:D	555	\pdf_startthread:D	618
\pdf_imagehicolor:D	552	\pdf_strcmp:D	585,
\pdf_imageresolution:D	539		646, 1288, 1294, 2487, 2497, 7223, 7234
\pdf_includechars:D	578	\pdf_textrbanner:D	571
\pdf_inclusionerrorlevel:D	550	\pdf_texrevision:D	570
\pdf_info:D	621	\pdf_texversion:D	593
\pdf_lastannot:D	598	\pdf_thread:D	617
\pdf_lastdemerits:D	601	\pdf_threadmargin:D	563
\pdf_lastobj:D	594	\pdf_tracingfonts:D	541
\pdf_lastxform:D	595	\pdf_trailer:D	627
\pdf_lastximage:D	596	\pdf_unescapehex:D	584
\pdf_lastximagepages:D	597	\pdf_uniformdeviate:D	586
\pdf_lastxpos:D	599	\pdf_uniqueresname:D	542
\pdf_lastypos:D	600	\pdf_vorigin:D	558
\pdf_leftmarginkern:D	579	\pdf_xform:D	608
\pdf_linkmargin:D	561	\pdf_xformname:D	574
\pdf_literal:D	630	\pdf_ximage:D	610
\pdf_lpcode:D	546	\pdfadjustspacing	543

<code>\pdfannot</code>	612	<code>\pdfoptionpdfinclusionerrorlevel</code> ..	551
<code>\pdfcatalog</code>	622	<code>\pdfoutline</code>	615
<code>\pdfcompresslevel</code>	537	<code>\pdfoutput</code>	535
<code>\pdfcreationdate</code>	572	<code>\pdfpageattr</code>	566
<code>\pdfdecimaldigits</code>	538	<code>\pdfpageheight</code>	560
<code>\pdfdest</code>	616	<code>\pdfpageref</code>	573
<code>\pdfdestmargin</code>	562	<code>\pdfpageresources</code>	567
<code>\pdfelapsedtime</code>	602	<code>\pdfpagesattr</code>	565
<code>\pdfendlink</code>	614	<code>\pdfpagewidth</code>	559
<code>\pdfendthread</code>	619	<code>\pdfpkmode</code>	568
<code>\pdfescapehex</code>	583	<code>\pdfpkresolution</code>	540
<code>\pdfescapeiname</code>	582	<code>\pdfprotrudechars</code>	544
<code>\pdfescapestring</code>	581	<code>\pdfrandomseed</code>	603
<code>\pdffiledump</code>	591	<code>\pdfrefobj</code>	607
<code>\pdffilemoddate</code>	589	<code>\pdfrefxform</code>	609
<code>\pdffilesize</code>	590	<code>\pdfrefximage</code>	611
<code>\pdffontattr</code>	626	<code>\pdfresettimer</code>	632
<code>\pdffontexpand</code>	628	<code>\pdfsavepos</code>	620
<code>\pdffontname</code>	575	<code>\pdfsetrandomseed</code>	633
<code>\pdffontobjnum</code>	576	<code>\pdfshellescape</code>	604
<code>\pdffontsize</code>	577	<code>\pdfstartlink</code>	613
<code>\pdfforcepagebox</code>	548	<code>\pdfstartthread</code>	618
<code>\pdfgamma</code>	554	<code>\pdfstrcmp</code>	32, 49, 59, 64, 67, 81, 585
<code>\pdfhorigin</code>	557	<code>\pdftexbanner</code>	571
<code>\pdfimageapplygamma</code>	553	<code>\pdftexrevision</code>	570
<code>\pdfimagegamma</code>	555	<code>\pdftexversion</code>	593
<code>\pdfimagehicolor</code>	552	<code>\pdfthread</code>	617
<code>\pdfimageresolution</code>	539	<code>\pdfthreadmargin</code>	563
<code>\pdfincludechars</code>	578	<code>\pdftracingfonts</code>	541
<code>\pdfinclusionerrorlevel</code>	550	<code>\pdftrailer</code>	627
<code>\pdfinfo</code>	621	<code>\pdfunescapehex</code>	584
<code>\pdflastannot</code>	598	<code>\pdfuniformdeviate</code>	586
<code>\pdflastdemerits</code>	601	<code>\pdfuniqueresname</code>	542
<code>\pdflastobj</code>	594	<code>\pdfvorigin</code>	558
<code>\pdflastxform</code>	595	<code>\pdfxform</code>	608
<code>\pdflastximage</code>	596	<code>\pdfxformname</code>	574
<code>\pdflastximagepages</code>	597	<code>\pdfximage</code>	610
<code>\pdflastxpos</code>	599	<code>\peek_after:NN</code>	
<code>\pdflastypos</code>	600	53, 2869, 2869, 2883, 2898, 3016
<code>\pdflinkmargin</code>	561	<code>\peek_catcode:N</code>	54, 2975
<code>\pdfliteral</code>	630	<code>\peek_catcode_ignore_spaces:N</code>	54, 2980
<code>\pdfmapfile</code>	624	<code>\peek_catcode_remove:N</code>	54, 2985
<code>\pdfmapline</code>	625	<code>\peek_catcode_remove_ignore_spaces:N</code>	54, 2990
<code>\pdfmdfivesum</code>	588	54, 2995
<code>\pdfminorversion</code>	536	<code>\peek_charcode:N</code>	54, 2995
<code>\pdfnames</code>	623	<code>\peek_charcode_ignore_spaces:N</code>	54, 3000
<code>\pdfnoligatures</code>	634	54, 3005
<code>\pdfnormaldeviate</code>	587	<code>\peek_charcode_remove:N</code>	54, 3005
<code>\pdfobj</code>	606	<code>\peek_charcode_remove_ignore_spaces:N</code>	54, 3010
<code>\pdfoptionalwaysusepdfpagebox</code>	549	54, 3010

- \peek_def_aux:nnnn [2941](#), [2941](#),
[2955](#), [2960](#), [2965](#), [2970](#), [2975](#), [2980](#),
[2985](#), [2990](#), [2995](#), [3000](#), [3005](#), [3010](#)
- \peek_def_aux_ii:nnnn
. [2941](#), [2942](#)–[2944](#), [2946](#)
- \peek_execute_branches: [2949](#), [3023](#)
- \peek_execute_branches_catcode:
[55](#), [2910](#), [2917](#), [2979](#), [2982](#), [2989](#), [2992](#)
- \peek_execute_branches_charcode:
[55](#), [2910](#), [2924](#), [2999](#), [3002](#), [3009](#), [3012](#)
- \peek_execute_branches_charcode_aux:NN
. [2910](#), [2931](#), [2933](#)
- \peek_execute_branches_meaning:
[55](#), [2910](#), [2910](#), [2959](#), [2962](#), [2969](#), [2972](#)
- \peek_gafter:NN [53](#), [2869](#), [2870](#)
- \peek_ignore_spaces_aux:
. [247](#), [3015](#), [3015](#), [3020](#)
- \peek_ignore_spaces_execute_branches:
. [247](#), [2964](#), [2974](#), [2984](#),
[2994](#), [3004](#), [3014](#), [3015](#), [3016](#), [3018](#)
- \peek_meaning:NTF [54](#), [2955](#)
- \peek_meaning_ignore_spaces:NTF [54](#), [2960](#)
- \peek_meaning_remove:NTF [54](#), [2965](#)
- \peek_meaning_remove_ignore_spaces:NTF
. [54](#), [2970](#)
- \peek_tmp:w [247](#), [2875](#), [2875](#), [2908](#), [3021](#)
- \peek_token_generic:NNF [2888](#)
- \peek_token_generic:NNT [2885](#)
- \peek_token_generic:NNTF
. [54](#), [2877](#), [2877](#), [2886](#), [2889](#)
- \peek_token_remove_generic:NNF [2903](#)
- \peek_token_remove_generic:NNT [2900](#)
- \peek_token_remove_generic:NNTF
. [54](#), [2891](#), [2891](#), [2901](#), [2904](#)
- \penalty [440](#)
- \postdisplaypenalty [287](#)
- \predisplaydirection [531](#)
- \predisdisplaypenalty [286](#)
- \predisplaysize [285](#)
- \pref_global:D [23](#), [895](#), [895](#),
[1376](#), [1381](#), [2567](#), [2871](#), [3063](#), [3093](#),
[3099](#), [3111](#), [3129](#), [3135](#), [3810](#), [3825](#),
[3847](#), [3855](#), [3924](#), [3944](#), [3950](#), [3954](#),
[4042](#), [4044](#), [4046](#), [4240](#), [4630](#), [4631](#),
[4642](#), [4647](#), [4657](#), [4667](#), [4681](#), [4695](#),
[5450](#), [5452](#), [5457](#), [5465](#), [5500](#), [5504](#),
[5510](#), [5516](#), [5530](#), [5536](#), [5544](#), [5832](#)
- \pref_global_chk: [3061](#), [3091](#),
[3097](#), [3127](#), [3133](#), [3808](#), [3823](#), [3845](#),
[3853](#), [4238](#), [4656](#), [4666](#), [4680](#), [4694](#)
- \pref_long:D [23](#), [895](#), [896](#),
[901](#), [904](#), [913](#), [916](#), [921](#), [924](#), [933](#), [936](#)
- \pref_protected:D
. [23](#), [895](#), [897](#), [900](#), [903](#), [906](#), [907](#),
[909](#), [910](#), [913](#), [916](#), [927](#), [930](#), [933](#), [936](#)
- \pretolerance [366](#)
- \prevdepth [413](#)
- \prevgraf [372](#)
- \prg_case_dim:nnn [38](#), [2195](#), [2195](#)
- \prg_case_dim_aux:nnn
. [2195](#), [2196](#), [2199](#), [2203](#)
- \prg_case_int:nnn
. [38](#), [2185](#), [2185](#), [3175](#), [3179](#), [3375](#)
- \prg_case_int_aux:nnn
. [2185](#), [2186](#), [2189](#), [2193](#)
- \prg_case_str:nnn [38](#), [2205](#), [2205](#)
- \prg_case_str_aux:nnn
. [2205](#), [2206](#), [2209](#), [2213](#)
- \prg_case_tl:Nnn [38](#), [2215](#), [2215](#)
- \prg_case_tl_aux:NNn [2215](#), [2216](#), [2219](#), [2223](#)
- \prg_conditional_form_F:nnn [1129](#)
- \prg_conditional_form_p:nnn [1126](#)
- \prg_conditional_form_T:nnn [1128](#)
- \prg_conditional_form_TF:nnn [1127](#)
- \prg_define_quicksort:nnn [2226](#), [2226](#), [2301](#)
- \prg_do_nothing:
. [15](#), [1429](#), [1429](#), [4494](#), [4501](#),
[4528](#), [4534](#), [4617](#), [4646](#), [7229](#), [7240](#)
- \prg_end_case:nw
. [2192](#), [2202](#), [2212](#), [2222](#), [2225](#), [2225](#)
- \prg_generate_conditional_aux:nnNNnnnn
. [986](#), [992](#),
[998](#), [1004](#), [1010](#), [1016](#), [1023](#), [1030](#), [1042](#)
- \prg_generate_conditional_aux:nnw
. [1043](#), [1047](#), [1052](#)
- \prg_generate_conditional_parm_aux:nnNNnnnn
. [1042](#)
- \prg_generate_conditional_parm_aux:nw
. [1042](#)
- \prg_generate_F_form_count:Nnnnn
. [1083](#), [1099](#)
- \prg_generate_F_form_parm:Nnnnn
. [1054](#), [1070](#)
- \prg_generate_p_form_count:Nnnnn
. [1083](#), [1083](#)
- \prg_generate_p_form_parm:Nnnnn
. [1054](#), [1054](#)
- \prg_generate_T_form_count:Nnnnn
. [1083](#), [1091](#)

\prg_generate_T_form_parm:Nnnnn	2719, 2726, 2730, 2737, 2741, 2748,
1054, 1062	2759, 2763, 2770, 2777, 2784, 2792,
\prg_generate_TF_form_count:Nnnnn . .	2829, 2836, 2841, 2848, 2850, 2852,
1083, 1107	2854, 2856, 3526, 3530, 3534, 3538,
\prg_generate_TF_form_parm:Nnnnn . .	3542, 3546, 3550, 3555, 3641, 3645,
1054, 1078	3896, 3977, 3994, 3998, 4002, 4006,
\prg_get_count_aux:nn	4010, 4014, 4018, 4257, 4264, 4279,
.. 1009, 1015, 1022, 1029, 1040, 1040	4286, 4300, 4312, 4323, 4331, 4463,
\prg_get_parm_aux:nw	4472, 4569, 4574, 4592, 4723, 4731,
985, 991, 997, 1003, 1040, 1041	4988, 5112, 5376, 5426, 5429, 5440,
\prg_new_conditional:Nnn	5825, 6714, 7075, 10257, 10264,
34, 1008, 1014,	10302, 10315, 10319, 10323, 10327,
1863, 2482, 2486, 2493, 2496, 2629,	10338, 10342, 10357, 10371, 10388,
2633, 2637, 2641, 2645, 2649, 2653,	10397, 10405, 10419, 10428, 10436
2657, 2661, 2665, 2669, 2673, 2677,	
2681, 2685, 2692, 2695, 2714, 2721,	\prg_return_true: 33,
2728, 2739, 2750, 2761, 2772, 2779,	980, 980, 1174, 1190, 1209, 1291,
2786, 2826, 3892, 3975, 4310, 4321,	1295, 1416, 1422, 1426, 1554, 1863,
4722, 4729, 5425, 5428, 5439, 5823	1864, 1867, 1870, 1874, 2023, 2171,
\prg_new_conditional:Npnn . . 34, 984,	2484, 2490, 2494, 2500, 2631, 2635,
990, 1414, 1863, 3979, 4263, 4284,	2639, 2643, 2647, 2651, 2655, 2659,
4296, 4567, 4571, 4589, 10253, 10260	2663, 2667, 2671, 2675, 2679, 2683,
\prg_new_eq_conditional:NNn	2690, 2694, 2698, 2719, 2726, 2737,
34, 1034, 1037, 1863, 4971, 4973,	2748, 2759, 2770, 2777, 2784, 2792,
5097, 5098, 5105–5108, 5366–5371	2858, 3526, 3530, 3534, 3538, 3542,
\prg_new_map_functions:Nn	3546, 3550, 3555, 3641, 3645, 3896,
42, 2334, 2334, 5182	3977, 3994, 3998, 4002, 4006, 4010,
\prg_new_protected_conditional:Nnn .	4014, 4018, 4257, 4264, 4276, 4286,
34, 1008, 1028, 1863, 5109, 5372, 7071	4300, 4315, 4326, 4331, 4463, 4472,
\prg_new_protected_conditional:Npnn	4569, 4574, 4592, 4723, 4731, 4991,
34, 984, 1002, 1863,	5112, 5376, 5426, 5429, 5440, 5825,
4270, 4461, 4470, 4975, 10267, 10285	5826, 6712, 7076, 10255, 10262,
\prg_quicksort:n 41, 2301	10312, 10345, 10354, 10367, 10384,
\prg_quicksort_compare:nnTF	10392, 10402, 10415, 10423, 10433
41, 2302, 2303	\prg_set_conditional:Nnn
\prg_quicksort_function:n 41, 2302, 2302	34, 1008, 1008, 1863, 6710
\prg_replicate:nn	\prg_set_conditional:Npnn
40, 1889, 1889, 7552, 7592, 7669	34, 984, 984, 1169, 1180, 1201,
\prg_replicate_ 1901	1287, 1293, 1297, 1421, 1425, 1552,
\prg_replicate_aux:N 1889, 1895, 1896, 1899	1863, 1863, 1866, 1869, 1872, 2022,
\prg_replicate_first_aux:N	2169, 3510, 3552, 3639, 3643, 4255
1889, 1891, 1898	\prg_set_eq_conditional:NNn
\prg_return_false: 33, 980,	34, 1034, 1034, 1863
982, 1171, 1176, 1188, 1193, 1211,	\prg_set_eq_conditional_aux:NNNn . .
1214, 1291, 1295, 1416, 1423, 1427,	1035, 1038, 1112, 1112
1554, 1863, 1864, 1867, 1870, 1874,	\prg_set_eq_conditional_aux:NNNw . .
2023, 2171, 2484, 2490, 2494, 2500,	1112, 1113, 1115, 1124
2631, 2635, 2639, 2643, 2647, 2651,	\prg_set_map_functions:Nn
2655, 2659, 2663, 2667, 2671, 2675,	42, 2334, 2336, 2342
2679, 2683, 2690, 2694, 2698, 2700,	\prg_set_protected_conditional:Nnn .
	34, 1008, 1021, 1863

- \prg_set_protected_conditional:Npnn [34](#), [984](#), [996](#), [1863](#)
- \prg_stepwise_function:nnnN [40](#), [1930](#), [1930](#)
- \prg_stepwise_function_decr:nnnN ... [1930](#), [1932](#), [1945](#), [1949](#)
- \prg_stepwise_function_incr:nnnN ... [1930](#), [1933](#), [1936](#), [1940](#)
- \prg_stepwise_inline:nnnn [40](#), [1954](#), [1954](#), [10630](#), [10635](#)
- \prg_stepwise_inline_decr:Nnnn [1958](#), [1972](#), [1976](#)
- \prg_stepwise_inline_decr:nnnn .. [1954](#)
- \prg_stepwise_inline_incr:Nnnn [1959](#), [1964](#), [1968](#)
- \prg_stepwise_inline_incr:nnnn .. [1954](#)
- \prg_stepwise_variable:nnnN [41](#), [1980](#), [1980](#)
- \prg_stepwise_variable_decr:nnnN .. [1980](#), [1982](#), [1994](#), [1998](#)
- \prg_stepwise_variable_incr:nnnN .. [1980](#), [1983](#), [1986](#), [1990](#)
- \prg_variable_get_scope:N [42](#), [2304](#), [2309](#)
- \prg_variable_get_scope_aux:w [2304](#), [2310](#), [2313](#)
- \prg_variable_get_type:N . [42](#), [2304](#), [2322](#)
- \prg_variable_get_type:w [2304](#)
- \prg_variable_get_type_aux:w [2326](#), [2330](#)
- \ProcessOptions [136](#)
- \prop_clear:c [108](#), [5272](#), [5273](#), [6002](#)
- \prop_clear:N [108](#), [5272](#), [5272](#), [5328](#)
- \prop_del:Nn [109](#), [5348](#), [5348](#), [5359](#)
- \prop_del:NV [109](#), [5348](#)
- \prop_del_aux:w [111](#), [5348](#), [5349](#), [5351](#), [5352](#)
- \prop_display:c [110](#), [5286](#)
- \prop_display:N [110](#), [5286](#), [5286](#), [5302](#), [5770](#), [5773](#)
- \prop_gclear:c [108](#), [5272](#), [5275](#)
- \prop_gclear:N [108](#), [5272](#), [5274](#), [5334](#)
- \prop_gdel:Nn [109](#), [5348](#), [5350](#), [5360](#)
- \prop_gdel:NV [109](#), [5348](#), [5751](#), [5763](#)
- \prop_get:cnN [109](#), [5309](#), [6065](#)
- \prop_get:cVN [109](#), [5309](#)
- \prop_get:NnN [109](#), [5309](#), [5309](#), [5313](#)
- \prop_get:NVN [109](#), [5309](#)
- \prop_get_aux:w ... [111](#), [5309](#), [5310](#), [5312](#)
- \prop_get_del_aux:w [111](#), [5318](#), [5319](#), [5320](#)
- \prop_get_gdel:NnN [109](#), [5318](#), [5318](#)
- \prop_gget:cnN [109](#)
- \prop_gget:cVN [109](#)
- \prop_gget:NnN [109](#), [5314](#), [5314](#), [5317](#)
- \prop_gget:NVN [109](#), [5314](#)
- \prop_gget_aux:w .. [111](#), [5314](#), [5315](#), [5316](#)
- \prop_gput:ccx [108](#), [5326](#), [6252](#)
- \prop_gput:cnN [108](#), [5326](#)
- \prop_gput:Nnn [108](#), [5326](#), [5332](#), [5347](#)
- \prop_gput:Nno [108](#), [5326](#)
- \prop_gput:NnV [108](#), [5326](#)
- \prop_gput:Nnx [108](#), [5326](#)
- \prop_gput:NVn [108](#), [5326](#), [5633](#), [5646](#)
- \prop_gput_if_new:Nnn .. [108](#), [5362](#), [5362](#)
- \prop_gset_eq:cc [109](#), [5276](#), [5283](#)
- \prop_gset_eq:cN [109](#), [5276](#), [5282](#)
- \prop_gset_eq:Nc [109](#), [5276](#), [5281](#)
- \prop_gset_eq:NN [109](#), [5276](#), [5280](#)
- \prop_if_empty:c [5367](#)
- \prop_if_empty:cTF [110](#), [5366](#)
- \prop_if_empty:N [5366](#)
- \prop_if_empty:NTF [110](#), [5366](#)
- \prop_if_empty_p:c [110](#), [5366](#)
- \prop_if_empty_p:N [110](#), [5366](#)
- \prop_if_eq:cc [5371](#)
- \prop_if_eq:ccTF [111](#), [5368](#)
- \prop_if_eq:cN [5369](#)
- \prop_if_eq:cNTF [111](#), [5368](#)
- \prop_if_eq:Nc [5370](#)
- \prop_if_eq:NcTF [111](#), [5368](#)
- \prop_if_eq:NN [5368](#)
- \prop_if_eq:NNTF [111](#), [5368](#)
- \prop_if_eq_p:cc [111](#), [5368](#)
- \prop_if_eq_p:cN [111](#), [5368](#)
- \prop_if_eq_p:Nc [111](#), [5368](#)
- \prop_if_eq_p:NN [111](#), [5368](#)
- \prop_if_in:ccTF [111](#), [5372](#)
- \prop_if_in:cnTF .. [111](#), [5372](#), [6054](#), [6057](#)
- \prop_if_in:Nn [5372](#)
- \prop_if_in:NnF [5380](#), [5652](#), [5659](#)
- \prop_if_in:NnT [5379](#)
- \prop_if_in:NnTF .. [111](#), [5372](#), [5378](#), [6047](#)
- \prop_if_in:NoTF [111](#), [5372](#)
- \prop_if_in:NVT [5697](#), [5737](#)
- \prop_if_in:NVTF [111](#), [5372](#)
- \prop_if_in_aux:w . [111](#), [5372](#), [5373](#), [5375](#)
- \prop_map_break: .. [110](#), [5387](#), [5403](#), [5403](#)
- \prop_map_function:cc [109](#), [5381](#)
- \prop_map_function:cN [109](#), [5381](#)
- \prop_map_function:Nc .. [109](#), [5381](#), [5398](#)
- \prop_map_function:NN [109](#), [5381](#), [5381](#), [5392](#)
- \prop_map_function_aux:w [111](#), [5381](#), [5382](#), [5385](#), [5390](#)

- \prop_map_inline:cn 110, 5393
 - \prop_map_inline:Nn 110, 5290, 5393, 5394, 5402
 - \prop_new:c ... 108, 5270, 5271, 5998, 6286
 - \prop_new:N 108, 5270, 5270, 5587, 5588, 5890, 5891, 6247, 6248
 - \prop_put:cnn 108, 5326, 6128, 6131
 - \prop_put:cnx 108, 5326
 - \prop_put:Nnn 108, 5326, 5326, 5346, 5591–5594, 6134
 - \prop_put:Nno 108, 5326
 - \prop_put:NnV 108, 5326
 - \prop_put:Nnx 108, 5326
 - \prop_put:NVn 108, 5326
 - \prop_put:NVV 108, 5326
 - \prop_put_aux:w 111, 5326, 5329, 5335, 5338
 - \prop_put_if_new_aux:w 111, 5362, 5363, 5364
 - \prop_set_eq:cc 109, 5276, 5279
 - \prop_set_eq:cN 109, 5276, 5278
 - \prop_set_eq:Nc 109, 5276, 5277
 - \prop_set_eq:NN 109, 5276, 5276
 - \prop_show:c 110, 5284, 5285
 - \prop_show:N 110, 5284, 5284
 - \prop_split_aux:Nnn 111, 5303, 5303, 5310, 5315, 5319, 5327, 5333, 5349, 5351, 5363, 5373
 - \prop_tmp:w 5304, 5307, 5323, 5324, 5342, 5343, 5353–5356
 - \protect 64, 834
 - \protected 96, 533
 - \ProvidesClass 753
 - \ProvidesExplClass 6, 747, 752
 - \ProvidesExplFile 6, 747, 756
 - \ProvidesExplPackage 6, 747, 748, 854, 1570, 1858, 2420, 2512, 3027, 3785, 4051, 4595, 4749, 5074, 5264, 5406, 5558, 5836, 6242, 6351, 6550, 7006, 7118, 10501
 - \ProvidesFile 757
 - \ProvidesPackage 749, 800
- Q**
- \Q 6405
 - \q 1148, 1892, 2125, 2129
 - \q_error 45, 2429, 2429
 - \q_mark 45, 2429, 2430, 3516, 3518, 3985, 3987
 - \q_nil 45, 960, 963, 2229, 2233, 2426, 2428, 2494, 2498, 3415, 3496, 4285, 4492, 4503, 4522, 4536, 4675, 5136, 5143, 5383, 5386, 6263, 6270, 6406, 6413, 6421, 6430, 6435, 6443, 6450, 6460, 6466, 6467, 6477, 6483, 6488, 6501, 6506, 6510, 6513
 - \q_no_value 45, 1779, 2148, 2426, 2427, 2483, 2488, 4465, 4474, 4492, 4503, 4536, 5111, 5114, 5307, 5323, 5342, 5355, 5383, 6442, 6450, 6456, 6461, 6477, 6484
 - \q_prop 111, 5269, 5269, 5304, 5307, 5323, 5339, 5342, 5355, 5365, 5383, 5385, 6259, 6261, 6278
 - \q_recursion_stop 44, 962, 965, 1045, 1113, 1724, 2187, 2197, 2207, 2217, 2225, 2350, 2358, 2388, 2411, 2431, 2432, 2454, 2455, 2467, 2468, 2478, 4341, 4345, 4358, 4367, 4372, 4383, 5191
 - \q_recursion_tail 44, 2187, 2197, 2207, 2217, 2350, 2358, 2388, 2411, 2431, 2431, 2435, 2441, 2455, 2468, 2478, 4341, 4345, 4358, 4367, 4372, 4383, 5191
 - \q_stop .. 45, 772, 776, 961, 964, 1148, 1150, 1157, 1159, 1441, 1445, 1771, 1776, 2148, 2150, 2229, 2233, 2295, 2311, 2313, 2324, 2326, 2330, 2426, 2426, 2455, 2468, 2478, 2686, 2688, 2716, 2718, 2723, 2725, 2733, 2736, 2744, 2747, 2755, 2758, 2766, 2769, 2774, 2776, 2781, 2783, 2788, 2791, 2804, 2810, 2816, 2822, 3305, 3341, 3512, 3514, 3516, 3524, 3528, 3532, 3536, 3540, 3544, 3548, 3981, 3983, 3985, 3992, 3996, 4000, 4004, 4008, 4012, 4016, 4297, 4299, 4317, 4327, 4330, 4462, 4465, 4471, 4474, 4487, 4503, 4518, 4522, 4524, 4528, 4536, 4553, 4555, 4557, 4559, 4561, 4562, 4568, 4573, 4577, 4591, 4676, 4685, 4890, 4893, 4903, 4906, 5110, 5114, 5130, 5133, 5136, 5138, 5304, 5307, 5383, 6413, 6478, 6740, 6742, 6747, 7201, 7203, 7220, 7224, 7229, 7235, 7240, 7248, 7253, 7255, 7256, 7261, 7264, 7269, 7302, 7307, 7529, 7532, 7547, 7549, 7550, 7555, 7561, 7563, 7565, 7567, 7568, 7571, 7574, 7577, 7580, 7583, 7586, 7589, 7590, 7598, 7601, 7610, 7612, 7622, 7628, 7630,

7631, 7633, 7636, 7639, 7642, 7645, 7648, 7652, 7656, 7659, 7667, 7672, 7675, 7684, 7686, 7692, 7694, 7695, 7697, 7702, 7706, 7710, 7714, 7718, 7722, 7726, 7730, 7739, 7743, 7744, 7756, 7758, 7777, 7797, 8276, 8281, 8394, 8445, 8616, 8621, 8627, 8630	\quark_if_nil:N 2493	\radical 261
\quark_if_nil:n 2496	\quark_if_nil:nF 2506, 2510	\raise 401
\quark_if_nil:nT . 2236, 2240, 2505, 2509	\quark_if_nil:NTF 44, 2493, 3418, 3499, 5141, 6496	\read 208
\quark_if_nil:nTF 2253, 2262, 2271, 2496, 2504, 2508	\quark_if_nil:oF 6458	\readline 483
\quark_if_nil:oTF 44, 2496	\quark_if_nil:VTF 44, 2496	\relax 2–5, 8, 12, 52, 58, 62, 78, 97, 100–109, 112–121, 243, 809
\quark_if_nil_p:N 44, 2493	\quark_if_nil_p:n .. 44, 2496, 2503, 2507	\relpenalty 304
\quark_if_nil_p:o 44, 2496	\quark_if_nil_p:V 44, 2496	\RequirePackage 47, 48, 821
\quark_if_no_value:N 2482	\quark_if_no_value:n 2486	\reverse_if:N 8, 860, 865
\quark_if_no_value:nF 5322, 5354	\quark_if_no_value:nF 4489	\right 302
\quark_if_no_value:NTF 10577	\quark_if_no_value:NTF .. 43, 2151, 2482	\righthypenmin 358
\quark_if_no_value:nTF 43, 2482, 4463, 4472, 4520, 5376	\quark_if_no_value_p:N 43, 2482	\rightmargin kern 580
\quark_if_no_value_p:n 43, 2482	\quark_if_recursion_tail_aux:w 2448, 2454, 2467, 2477	\rightskip 360
\quark_if_recursion_tail_stop:N 44, 2433, 2433, 4378, 5198	\quark_if_recursion_tail_stop:n 44, 2363, 2448, 2448, 2480, 4348	\romannumeral 435
\quark_if_recursion_tail_stop:o 44, 2448	\quark_if_recursion_tail_stop:Nn 45, 2220, 2433, 2439	\rpcode 547
\quark_if_recursion_tail_stop_do:Nn 45, 2190, 2200, 2210, 2448, 2461, 2481, 4386	\quark_if_recursion_tail_stop_do:nn 45, 2448	
\quark_if_recursion_tail_stop_do:on 45, 2448	\quark_new:N ... 43, 2425, 2425–2432, 5269	
		S
		\S 2710
		\savecatcodetable 640
		\savinghyphcodes 522
		\savingvdiscards 523
		\scan_align_safe_stop: 40, 1873, 1879, 1879
		\scan_stop: 25, 841, 891, 891, 1116, 1142, 1757–1765, 1885, 2693, 2811, 2817, 2823, 3558, 3565, 3572, 3682, 3691, 3692, 3709, 3801, 3816, 3831, 3838, 3861, 3866, 3873, 3906, 3907, 3921, 3947, 3952, 3969, 3976, 4041, 4043, 4045, 4410–4413, 4737, 5120, 5474, 5477, 5480, 5496, 5497, 5527, 5634, 5647, 5929, 5930, 7209–7211, 7250, 7262, 7270, 7275, 7309, 7310, 7326, 7334, 7370, 7379, 7393, 7402, 7476, 7664, 7768, 7787, 7971, 7982, 7983, 8086, 8123, 8127, 8138, 8142, 8155, 8159, 8207, 8267, 8271, 8277–8279, 8286, 8296, 8350, 8381, 8441, 8449, 8450, 8453–8455, 8468–8470, 8484, 8485, 8522, 8530, 8571, 8585, 8589, 8594, 8601, 8623, 8624, 8632, 8633, 8648, 8656, 8664, 8678, 8691, 8703, 8743, 8745, 8747, 8767, 8771, 8775, 8904, 8992, 9046, 9251, 9270, 9450, 9460, 9500, 9502, 9532, 9574–9577, 9591, 9597, 9601, 9638, 9825, 9846, 9886, 9894, 9902, 9926, 9979, 9981, 9983, 10016, 10024, 10200, 10227, 10229, 10231, 10233, 10235, 10579
		\scantokens 481
		\scriptfont 427
	R	
\R 1760, 2710, 2800		

<code>\scriptscriptfont</code>	428	<code>\seq_gpush:Nn</code>	99, 4867, 4877
<code>\scriptscriptstyle</code>	273	<code>\seq_gpush:No</code>	99, 4867, 4880
<code>\scriptspace</code>	313	<code>\seq_gpush:Nv</code>	99, 4867, 4878, 7089
<code>\scriptstyle</code>	272	<code>\seq_gpush:Nv</code>	99, 4867, 4879
<code>\scrollmode</code>	238	<code>\seq_gpush:Nx</code>	99, 4867, 4881, 10566
<code>\seq_break:</code>		<code>\seq_gput_left:cn</code>	93, 4792, 4882
324, 4800, 4800, 4802, 4809, 4928, 4960		<code>\seq_gput_left:co</code>	93, 4792, 4885
<code>\seq_break:n</code>		<code>\seq_gput_left:cV</code>	93, 4792, 4883
... 324, 4800, 4801, 4803, 4988, 4991		<code>\seq_gput_left:cv</code>	93, 4792, 4884
<code>\seq_break_point:n</code>	324,	<code>\seq_gput_left:cx</code>	93, 4792, 4886
4800, 4801, 4804, 4804, 4816, 4852,		<code>\seq_gput_left:Nn</code>	
4863, 4891, 4904, 4917, 4945, 4989		... 93, 4792, 4792, 4796, 4797, 4877	
<code>\seq_clear:c</code>	91, 4762, 4763	<code>\seq_gput_left:No</code>	93, 4792, 4880
<code>\seq_clear:N</code>	91, 4762, 4762, 5005	<code>\seq_gput_left:Nv</code>	93, 4792, 4878
<code>\seq_clear_new:c</code>	91, 4766, 4767	<code>\seq_gput_left:Nv</code>	93, 4792, 4879
<code>\seq_clear_new:N</code>	91, 4766, 4766	<code>\seq_gput_left:Nx</code>	93, 4792, 4881
<code>\seq_concat:ccc</code>	92, 4778	<code>\seq_gput_right:cn</code>	93, 4792
<code>\seq_concat:NNN</code>	92, 4778, 4778, 4782	<code>\seq_gput_right:co</code>	93, 4792
<code>\seq_display:c</code>	5070, 5071	<code>\seq_gput_right:cV</code>	93, 4792
<code>\seq_display:N</code>	5070, 5070	<code>\seq_gput_right:cv</code>	93, 4792
<code>\seq_gclear:c</code>	91, 4762, 4765	<code>\seq_gput_right:cx</code>	93, 4792
<code>\seq_gclear:N</code>	91, 4762, 4764	<code>\seq_gput_right:Nn</code>	
<code>\seq_gclear_new:c</code>	91, 4766, 4769	... 93, 4792, 4794, 4798, 4799	
<code>\seq_gclear_new:N</code>	91, 4766, 4768	<code>\seq_gput_right:No</code>	93, 4792
<code>\seq_gconcat:ccc</code>	92, 4778	<code>\seq_gput_right:Nv</code>	93, 4792, 7027
<code>\seq_gconcat:NNN</code>	92, 4778, 4780, 4783	<code>\seq_gput_right:Nv</code>	93, 4792
<code>\seq_get:cN</code>	98, 4965, 4966	<code>\seq_gput_right:Nx</code>	93, 4792, 7084
<code>\seq_get:NN</code>	98, 4965, 4965	<code>\seq_gremove_all:cn</code>	95, 5015
<code>\seq_get_left:cN</code>	94, 4887, 4966, 5069	<code>\seq_gremove_all:Nn</code>	95, 5015, 5017, 5040
<code>\seq_get_left:NN</code>		<code>\seq_gremove_duplicates:c</code>	95, 4999
... 94, 4887, 4887, 4895, 4965, 5068		<code>\seq_gremove_duplicates:N</code>	
<code>\seq_get_left_aux:Nw</code>	4887, 4890, 4893	... 95, 4999, 5001, 5014	
<code>\seq_get_right:cN</code>	94, 4913	<code>\seq_gset_eq:cc</code>	92, 4770, 4777
<code>\seq_get_right:NN</code>	94, 4913, 4913, 4936	<code>\seq_gset_eq:cN</code>	92, 4770, 4776
<code>\seq_get_right_aux:NN</code>	4913, 4916, 4919	<code>\seq_gset_eq:Nc</code>	92, 4770, 4775
<code>\seq_get_right_aux:NNN</code>	4937	<code>\seq_gset_eq:NN</code>	92, 4770, 4774, 5002
<code>\seq_get_right_aux_ii:NNN</code>	4937	<code>\seq_if_empty:c</code>	4973
<code>\seq_get_right_loop:nn</code>		<code>\seq_if_empty:cTF</code>	96, 4971
... 4913, 4922, 4931, 4934, 4951		<code>\seq_if_empty:N</code>	4971
<code>\seq_gpop:cN</code>	98, 4965, 4970	<code>\seq_if_empty:NTF</code>	96, 4971, 5044
<code>\seq_gpop:NN</code>	98, 4965, 4969, 7092, 10576	<code>\seq_if_empty_err_break:N</code>	
<code>\seq_gpop_left:cN</code>	94, 4896, 4970	324, 4805, 4805, 4889, 4902, 4915, 4943	
<code>\seq_gpop_left:NN</code>	94, 4896, 4898, 4912, 4969	<code>\seq_if_empty_p:c</code>	96, 4971
<code>\seq_gpop_right:cN</code>	94, 4937	<code>\seq_if_empty_p:N</code>	96, 4971
<code>\seq_gpop_right:NN</code>	94, 4937, 4939, 4964	<code>\seq_if_in:cnTF</code>	96, 4975
<code>\seq_gpush:cn</code>	99, 4867, 4882	<code>\seq_if_in:coTF</code>	96, 4975
<code>\seq_gpush:co</code>	99, 4867, 4885	<code>\seq_if_in:cVTF</code>	96, 4975
<code>\seq_gpush:cV</code>	99, 4867, 4883	<code>\seq_if_in:cvTF</code>	96, 4975
<code>\seq_gpush:cv</code>	99, 4867, 4884	<code>\seq_if_in:cxTF</code>	96, 4975
<code>\seq_gpush:cx</code>	99, 4867, 4886	<code>\seq_if_in:Nn</code>	4975

- \seq_if_in:NnF ... [4994](#), [4995](#), [5008](#), [7097](#)
- \seq_if_in:NnT ... [4992](#), [4993](#)
- \seq_if_in:NnTF ... [96](#), [4975](#), [4996](#), [4997](#)
- \seq_if_in:NoTF ... [96](#), [4975](#)
- \seq_if_in:NVTF ... [96](#), [4975](#)
- \seq_if_in:NvTF ... [96](#), [4975](#)
- \seq_if_in:NxTF ... [96](#), [4975](#)
- \seq_if_in_aux: ... [4975](#), [4984](#), [4991](#)
- \seq_item:n ... [324](#), [4753](#), [4753](#), [4785](#),
[4787](#), [4793](#), [4795](#), [4829](#), [4834](#), [4839](#),
[4845](#), [4893](#), [4906](#), [4949](#), [4980](#), [5033](#)
- \seq_map_break: [97](#), [4802](#), [4802](#), [4815](#), [7062](#)
- \seq_map_break:n ... [97](#), [4802](#), [4803](#)
- \seq_map_function:cN ... [96](#), [4812](#)
- \seq_map_function:NN ...
... [96](#), [4812](#), [4812](#), [4824](#), [5056](#)
- \seq_map_function_aux:NNn ...
... [4812](#), [4814](#), [4818](#), [4822](#)
- \seq_map_inline:cn ... [96](#), [4848](#)
- \seq_map_inline:Nn ...
[96](#), [4848](#), [4848](#), [4854](#), [5006](#), [7056](#), [7106](#)
- \seq_map_variable:ccn ... [97](#), [4855](#)
- \seq_map_variable:cNn ... [97](#), [4855](#)
- \seq_map_variable:Ncn ... [97](#), [4855](#)
- \seq_map_variable:NNn ...
... [97](#), [4855](#), [4855](#), [4865](#), [4866](#)
- \seq_new:c ... [4](#), [91](#), [4760](#), [4761](#)
- \seq_new:N ... [4](#), [91](#), [4760](#), [4760](#),
[4998](#), [7021](#), [7022](#), [7031](#), [7033](#), [10533](#)
- \seq_pop:cN ... [98](#), [4965](#), [4968](#)
- \seq_pop:NN ... [98](#), [4965](#), [4967](#)
- \seq_pop_item_def: ... [324](#), [4826](#),
[4842](#), [4852](#), [4863](#), [4927](#), [4959](#), [5037](#)
- \seq_pop_left:cN ... [94](#), [4896](#), [4968](#)
- \seq_pop_left:NN [94](#), [4896](#), [4896](#), [4911](#), [4967](#)
- \seq_pop_left_aux:NNN ...
... [4896](#), [4897](#), [4899](#), [4900](#)
- \seq_pop_left_aux:Nw ... [4896](#), [4903](#), [4906](#)
- \seq_pop_right:cN ... [94](#), [4937](#)
- \seq_pop_right:NN .. [94](#), [4937](#), [4937](#), [4963](#)
- \seq_pop_right_aux:NNN .. [4938](#), [4940](#), [4941](#)
- \seq_pop_right_aux_ii:NNN ... [4944](#), [4947](#)
- \seq_push:cn ... [99](#), [4867](#), [4872](#)
- \seq_push:co ... [99](#), [4867](#), [4875](#)
- \seq_push:cV ... [99](#), [4867](#), [4873](#)
- \seq_push:cv ... [99](#), [4874](#)
- \seq_push:cx ... [99](#), [4867](#), [4876](#)
- \seq_push:Nn ... [99](#), [4867](#), [4867](#)
- \seq_push:No ... [99](#), [4867](#), [4870](#)
- \seq_push:Nv ... [99](#), [4867](#), [4868](#)
- \seq_push:Nv ... [99](#), [4867](#), [4869](#)
- \seq_push:Nx ... [99](#), [4867](#), [4871](#)
- \seq_push_item_def:n ...
[324](#), [4826](#), [4826](#), [4850](#), [4921](#), [4949](#), [5021](#)
- \seq_push_item_def:x [324](#), [4826](#), [4831](#), [4857](#)
- \seq_push_item_def_aux: ...
... [4826](#), [4828](#), [4833](#), [4836](#)
- \seq_put_left:cn ... [92](#), [4784](#), [4872](#)
- \seq_put_left:co ... [92](#), [4784](#), [4875](#)
- \seq_put_left:cV ... [92](#), [4784](#), [4873](#)
- \seq_put_left:cv ... [92](#), [4784](#), [4874](#)
- \seq_put_left:cx ... [92](#), [4784](#), [4876](#)
- \seq_put_left:Nn ...
... [92](#), [4784](#), [4788](#), [4789](#), [4867](#)
- \seq_put_left:No ... [92](#), [4784](#), [4870](#)
- \seq_put_left:Nv ... [92](#), [4784](#), [4868](#)
- \seq_put_left:Nx ... [92](#), [4784](#), [4871](#)
- \seq_put_right:cn ... [93](#), [4784](#)
- \seq_put_right:co ... [93](#), [4784](#)
- \seq_put_right:cV ... [93](#), [4784](#)
- \seq_put_right:cv ... [93](#), [4784](#)
- \seq_put_right:cx ... [93](#), [4784](#)
- \seq_put_right:Nn ...
... [93](#), [4738](#), [4784](#), [4786](#), [4790](#),
[4791](#), [5009](#), [5413](#), [7053](#), [7098](#), [7113](#)
- \seq_put_right:No ... [93](#), [4784](#)
- \seq_put_right:Nv ... [93](#), [4784](#)
- \seq_put_right:Nx ... [93](#), [4784](#)
- \seq_remove_all:cn ... [95](#), [5015](#)
- \seq_remove_all:Nn ...
... [95](#), [5015](#), [5015](#), [5039](#), [7101](#)
- \seq_remove_all_aux:NNn ...
... [5015](#), [5016](#), [5018](#), [5019](#)
- \seq_remove_duplicates:c ... [95](#), [4999](#)
- \seq_remove_duplicates:N ...
... [95](#), [4999](#), [4999](#), [5013](#), [7104](#)
- \seq_remove_duplicates_aux:NN ...
... [4999](#), [5000](#), [5002](#), [5003](#)
- \seq_set_eq:cc ... [91](#), [4770](#), [4773](#)
- \seq_set_eq:cN ... [91](#), [4770](#), [4772](#)
- \seq_set_eq:Nc ... [91](#), [4770](#), [4771](#)
- \seq_set_eq:NN ...
... [91](#), [4770](#), [4770](#), [5000](#), [7051](#), [7067](#)
- \seq_show:c ... [99](#), [5042](#), [5071](#)
- \seq_show:N ... [99](#), [5042](#), [5042](#), [5067](#), [5070](#)
- \seq_show_aux:n ... [5042](#), [5056](#), [5061](#)
- \seq_show_aux:w ... [5042](#), [5058](#), [5066](#)
- \seq_top:cN ... [5068](#), [5069](#)

<code>\seq_top:NN</code>	5068, 5068	<code>\skip_show:c</code>	69, 3870
<code>\seq_use_error:</code>	4755	<code>\skip_show:N</code>	69, 3870, 3870, 3871
<code>\setbox</code>	409	<code>\skip_split_finite_else_action:nnNN</code>	70, 3898, 3898
<code>\setlabel</code>	6323, 6329, 6330, 6332, 6333, 6337, 6338, 6340, 6341	<code>\skip_sub:Nn</code>	69, 3830, 3837, 3856
<code>\setlanguage</code>	167	<code>\skip_use:c</code>	69, 3868
<code>\setname</code>	6314, 6329, 6330, 6332, 6333, 6337, 6340	<code>\skip_use:N</code>	69, 3868, 3868, 3869
<code>\sfcode</code>	464	<code>\skip_vertical:c</code>	70, 3858
<code>\shipout</code>	374	<code>\skip_vertical:N</code> 70, 3858, 3863, 3864, 3866	
<code>\show</code>	217	<code>\skip_vertical:n</code>	70, 3858, 3865
<code>\showbox</code>	219	<code>\skip_zero:c</code>	69, 3815
<code>\showboxbreadth</code>	233	<code>\skip_zero:N</code> ...	69, 3815, 3815, 3826, 3828
<code>\showboxdepth</code>	234	<code>\skipdef</code>	154
<code>\showgroups</code>	494	<code>\spacefactor</code>	373
<code>\showifs</code>	495	<code>\spaceskip</code>	368
<code>\showlists</code>	220	<code>\span</code>	181
<code>\showMemUsage</code>	842, 851	<code>\special</code>	443, 631
<code>\showthe</code>	218	<code>\splitbotmark</code>	252
<code>\showtokens</code>	482	<code>\splitbotmarks</code>	478
<code>\skewchar</code>	431	<code>\splitdiscards</code>	525
<code>\skip</code>	455	<code>\splitfirstmark</code>	251
<code>\skip@</code>	3876	<code>\splitfirstmarks</code>	477
<code>\skip_add:cn</code>	69, 3830	<code>\splitmaxdepth</code>	421
<code>\skip_add:Nn</code> ...	69, 3830, 3830, 3836, 3848	<code>\splittopskip</code>	422
<code>\skip_eval:n</code>	70, 3872, 3872	<code>\startrecording</code>	6304, 6311
<code>\skip_gadd:cn</code>	69, 3830	<code>\str_if_eq:nn</code>	1287
<code>\skip_gadd:Nn</code>	69, 3830, 3843, 3850	<code>\str_if_eq:nnF</code>	1846–1850, 5240
<code>.skip_gset:c</code>	139	<code>\str_if_eq:nnT</code>	1841–1845, 5023
<code>\skip_gset:cn</code>	69, 3800	<code>\str_if_eq:nnTF</code>	11, 1287, 1851–1855, 3309, 3312
<code>.skip_gset:N</code>	139	<code>\str_if_eq:noTF</code>	11, 1836
<code>\skip_gset:Nn</code>	69, 3800, 3806, 3814	<code>\str_if_eq:nVTF</code>	11, 1836
<code>\skip_gsub:Nn</code>	69, 3830, 3851	<code>\str_if_eq:onTF</code>	11, 1836
<code>\skip_gzero:c</code>	69, 3815	<code>\str_if_eq:VnTF</code>	11, 1836
<code>\skip_gzero:N</code>	69, 3815, 3821, 3829	<code>\str_if_eq:VVTF</code>	11, 1836
<code>\skip_horizontal:c</code>	70, 3858	<code>\str_if_eq:xx</code>	1293
<code>\skip_horizontal:N</code>	70, 3858, 3858, 3859, 3861	<code>\str_if_eq:xxTF</code>	11, 1287, 2211, 4730
<code>\skip_horizontal:n</code>	70, 3858, 3860	<code>\str_if_eq_p:nn</code>	11, 1199, 1287, 1298, 1836–1840
<code>\skip_if_infinite_glue:n</code>	3892	<code>\str_if_eq_p:no</code>	11, 1836
<code>\skip_if_infinite_glue:nTF</code> 70, 3892, 3899		<code>\str_if_eq_p:nV</code>	11, 1836
<code>\skip_if_infinite_glue_p:n</code>	70, 3892	<code>\str_if_eq_p:on</code>	11, 1836
<code>\skip_new:c</code>	68, 3790	<code>\str_if_eq_p:Vn</code>	11, 1836
<code>\skip_new:N</code>	68, 3790, 3794, 3799, 3877–3881, 3883, 3885, 7517	<code>\str_if_eq_p:VV</code>	11, 1836
<code>.skip_set:c</code>	139	<code>\str_if_eq_p:xx</code>	11, 1287
<code>\skip_set:cn</code>	69, 3800	<code>\strcmp</code>	59, 646
<code>.skip_set:N</code>	139	<code>\string</code>	67, 81, 436
<code>\skip_set:Nn</code>	69, 3800, 3800, 3811, 3813, 3884, 3886		

T

`\T`

1762, 2704

<code>\t</code>	2707	1143, 1640, 1641, 1759–1765, 2306,	
<code>\tabskip</code>	182	2319, 2517, 2521, 2526, 4412, 4413	
<code>\tex_above:D</code>	264	<code>\tex_char:D</code>	316
<code>\tex_abovedisplayshortskip:D</code>	277	<code>\tex_chardef:D</code> .	151, 1130, 1131, 1227–
<code>\tex_abovedisplayskip:D</code>	278	1231, 2158, 2160, 3709, 5603, 5607	
<code>\tex_abovewithdelims:D</code>	265	<code>\tex_cleaders:D</code>	334
<code>\tex_accent:D</code>	315	<code>\tex_closein:D</code>	210, 5762
<code>\tex_adjdemerits:D</code>	352	<code>\tex_closeout:D</code>	205, 5750
<code>\tex_advancd:D</code>	8604	<code>\tex_clubpenalty:D</code>	345
<code>\tex_advance:D</code>	159, 3040, 3831,	<code>\tex_copy:D</code>	402, 5446, 5487
3838, 3947, 3952, 4043, 4045, 7300,		<code>\tex_count:D</code>	453
7311, 7317, 7327, 7335, 7361, 7371,		<code>\tex_countdef:D</code>	152, 1224, 2729, 3045, 3691
7380, 7384, 7394, 7403, 7445, 7488,		<code>\tex_cr:D</code>	177
7901, 7938, 7963, 7970, 7976, 7999,		<code>\tex_crcr:D</code>	178
8015, 8043, 8130, 8131, 8144, 8145,		<code>\tex_csname:D</code>	
8274, 8282, 8289, 8392, 8422–8424,		..	240, 644, 808, 823, 884, 9571, 9951
8426, 8427, 8458, 8459, 8463, 8464,		<code>\tex_day:D</code>	448
8472, 8473, 8476, 8477, 8483, 8603,		<code>\tex_deadcycles:D</code>	382
8614, 8625, 8634, 8638, 8665, 8669,		<code>\tex_def:D</code>	147, 648, 654, 668, 681, 682,
8679, 8695, 8704, 8749, 8750, 8753,		686, 690, 695–697, 699, 701–707,	
8754, 8804, 8816, 9049, 9050, 9082,		710, 714, 716–719, 723, 730, 732–	
9087, 9090, 9091, 9095, 9096, 9101,		734, 738, 740–742, 748, 752, 756,	
9102, 9106, 9107, 9110, 9111, 9114,		776, 777, 784, 798, 800, 804, 809,	
9115, 9544, 9563, 9618, 9622, 9644,		815, 820, 821, 830, 887, 889, 890, 898	
9659, 9660, 9664, 9665, 9670, 9671,		<code>\tex_defaultthyphenchar:D</code>	432
9675, 9676, 9679, 9680, 9683, 9684,		<code>\tex_defaultskewchar:D</code>	433
9800, 9804, 9848, 9876, 9903, 9932,		<code>\tex_delcode:D</code>	463
9939, 9942, 9986, 9989, 9990, 9992,		<code>\tex_delimiter:D</code>	257
10006, 10026, 10029, 10042, 10043,		<code>\tex_delimiterfactor:D</code>	306
10046, 10047, 10052, 10053, 10246		<code>\tex_delimitershortfall:D</code>	305
<code>\tex_afterassignment:D</code>		<code>\tex_dimen:D</code>	454
.....	169, 2908, 3020, 7251	<code>\tex_dimendef:D</code>	153, 2751, 3911
<code>\tex_aftergroup:D</code>	170, 894	<code>\tex_discretionary:D</code>	317
<code>\tex_atop:D</code>	266	<code>\tex_displayindent:D</code>	282
<code>\tex_atopwithdelims:D</code>	267	<code>\tex_displaylimits:D</code>	292
<code>\tex_badness:D</code>	414	<code>\tex_displaystyle:D</code>	270
<code>\tex_baselineskip:D</code>	342	<code>\tex_displaywidowpenalty:D</code>	281
<code>\tex_batchmode:D</code>	235	<code>\tex_displaywidth:D</code>	283
<code>\tex_begingroup:D</code>	173, 641, 691, 801, 892	<code>\tex_divide:D</code>	
<code>\tex_belowdisplayshortskip:D</code>	279	160, 7328, 7372, 7395, 7975, 8254,	
<code>\tex_belowdisplayskip:D</code>	280	8442, 8506, 8549, 8593, 8596, 8598,	
<code>\tex_binoppenalty:D</code>	303	8600, 8658, 8696, 9896, 9944–9946	
<code>\tex_botmark:D</code>	250	<code>\tex_doublehyphenemerits:D</code>	350
<code>\tex_box:D</code>	458, 5448, 5485	<code>\tex_dp:D</code>	461, 5468
<code>\tex_boxmaxdepth:D</code>	420	<code>\tex_dump:D</code>	444
<code>\tex_brokenpenalty:D</code>	377	<code>\tex_edef:D</code> ...	148, 651, 760, 762, 767, 899
<code>\tex_catcode:D</code>	462,	<code>\tex_else:D</code>	201, 645,
656–658, 660–665, 669–671, 673–		650, 709, 726, 771, 780, 799, 863,	
678, 683, 684, 687, 688, 692, 818,		1781, 2443, 2472, 3560, 3567, 3574,	
		4277, 7206, 7233, 7281, 7291, 7340,	

7346, 7355, 7453, 7496, 7538, 7542, 7558, 7607, 7614, 7617, 7625, 7663, 7681, 7689, 7747, 7751, 7769, 7772, 7788, 7791, 7812, 7815, 7818, 7821, 7824, 7827, 7830, 7833, 7836, 7850, 7853, 7856, 7859, 7862, 7865, 7868, 7871, 7874, 7909, 7946, 7973, 7986, 7991, 8048, 8089, 8105, 8129, 8151, 8161, 8227, 8230, 8324, 8334, 8370, 8373, 8408, 8410, 8413, 8457, 8462, 8482, 8649, 8716, 8729, 8763, 8787, 8806, 8842, 8859, 8863, 8881, 8896, 8941, 8953, 8969, 8984, 9012, 9014, 9023, 9039, 9044, 9058, 9094, 9100, 9105, 9141, 9158, 9162, 9178, 9189, 9192, 9195, 9204, 9208, 9234, 9238, 9262, 9416, 9428, 9439, 9454, 9459, 9464, 9469, 9476, 9491, 9505, 9511, 9535, 9553, 9585, 9593, 9617, 9620, 9663, 9669, 9674, 9754, 9763, 9786, 9799, 9802, 9817, 9834, 9853, 9887, 9914, 9988, 9995, 10017, 10045, 10051, 10093, 10100, 10110, 10121, 10135, 10143, 10155, 10169, 10178, 10184, 10256, 10263, 10313, 10317, 10321, 10325, 10339, 10343, 10348, 10355, 10359, 10368, 10372, 10375, 10385, 10389, 10393, 10398, 10403, 10416, 10420, 10424, 10429, 10434	<code>\tex_everyvbox:D</code> 424 <code>\tex_exhyphenpenalty:D</code> 347 <code>\tex_expandafter:D</code> 171, 642, 644, 725, 727, 765, 768, 772, 807, 811, 822, 878, 1576, 3303, 3559, 3561, 3566, 3568, 3573, 3575, 7091, 7201, 7220, 7224, 7228, 7232, 7235, 7239, 7242, 7263, 7282, 7290, 7292, 7301, 7303, 7318, 7320, 7332, 7347, 7354, 7356, 7362, 7365, 7376, 7385, 7388, 7399, 7458, 7480, 7501, 7529, 7537, 7540, 7543, 7557, 7559, 7598, 7606, 7608, 7624, 7626, 7672, 7680, 7682, 7688, 7690, 7746, 7749, 7752, 7764, 7783, 7899, 7914, 7936, 7951, 7964, 8001, 8024, 8053, 8058, 8059, 8088, 8090, 8110, 8235, 8281, 8290, 8335, 8378, 8393, 8400, 8407, 8414–8416, 8615, 8617, 8626, 8639, 8642, 8670, 8673, 8683, 8686, 8700, 8715, 8721, 8728, 8734–8736, 8805, 8817, 8847, 8864, 8901, 8912, 8914, 8916, 8918, 8946, 8954, 8989, 9000, 9002, 9004, 9006, 9051, 9065, 9118, 9146, 9163, 9177, 9181, 9205, 9209, 9235, 9239, 9267, 9421, 9429, 9451–9453, 9455–9457, 9461–9463, 9470, 9475, 9481, 9490, 9494, 9507– 9509, 9513, 9514, 9526, 9569, 9635, 9687, 9752, 9760, 9769, 9779, 9787, 9822, 9840, 9877, 9880, 9913, 9915, 9927, 9934, 9949, 9994, 9996, 10007, 10010, 10056, 10111, 10120, 10132– 10134, 10154, 10163–10167, 10171– 10176, 10180–10182, 10185, 10197
<code>\tex_emergencystretch:D</code> 365 <code>\tex_end:D</code> 239, 787, 6081, 6178 <code>\tex_endcsname:D</code> 241, 644, 808, 823, 885, 8862, 8952, 9161, 9427, 9437, 9571, 9785, 9951 <code>\tex_endgroup:D</code> 174, 643, 700, 715, 805, 893 <code>\tex_endinput:D</code> 213 <code>\tex_endlinechar:D</code> 255, 659, 672, 4421, 4437, 4447, 4455 <code>\tex_eqno:D</code> 275 <code>\tex_errhelp:D</code> 221, 5942 <code>\tex_errmessage:D</code> . . . 215, 844, 1241, 5947 <code>\tex_errorcontextlines:D</code> 222, 5841 <code>\tex_errorstopmode:D</code> 236 <code>\tex_escapechar:D</code> 254 <code>\tex_everycr:D</code> 183 <code>\tex_everydisplay:D</code> 284, 789 <code>\tex_everyhbox:D</code> 423 <code>\tex_everyjob:D</code> 452, 7012, 7014, 7024, 7026 <code>\tex_everymath:D</code> 308, 788 <code>\tex_everypar:D</code> 371	<code>\tex_fam:D</code> 163 <code>\tex_fi:D</code> 202, 647, 666, 679, 711, 728, 773, 782, 826, 827, 864, 1136, 1140, 1783, 2437, 2445, 2459, 2474, 3562, 3569, 3576, 4280, 4810, 4950, 4954, 4985, 5025, 5028, 7208, 7244, 7245, 7278, 7283, 7293, 7304, 7321, 7345, 7348, 7357, 7366, 7389, 7455, 7498, 7535, 7545, 7546, 7560, 7604, 7609, 7619, 7620, 7627, 7665, 7678, 7683, 7691, 7754, 7755, 7771, 7775, 7790, 7795, 7814, 7817, 7820, 7823, 7826, 7829, 7832, 7835, 7838, 7852, 7855, 7858, 7861, 7864, 7867, 7870, 7873, 7876, 7900, 7911,

7937, 7948, 7977, 7988, 7996–7998, 8002, 8050, 8091, 8107, 8132, 8146, 8160, 8163, 8229, 8232, 8337, 8338, 8372, 8375, 8402, 8403, 8417–8419, 8428, 8460, 8465, 8474, 8478, 8486, 8487, 8605, 8618, 8643, 8657, 8674, 8714, 8722, 8737–8739, 8751, 8755, 8776, 8777, 8785, 8794, 8819, 8820, 8844, 8866, 8873, 8885, 8898, 8919, 8943, 8955, 8973, 8986, 9007, 9011, 9016, 9017, 9043, 9048, 9052, 9066, 9098, 9104, 9112, 9116, 9117, 9119, 9143, 9165, 9172, 9182, 9191, 9197, 9198, 9207, 9210, 9237, 9240, 9264, 9418, 9430, 9441, 9458, 9467, 9468, 9471, 9482, 9510, 9515, 9516, 9537, 9546, 9556, 9565, 9602, 9603, 9615, 9623, 9624, 9667, 9673, 9681, 9685, 9686, 9688, 9762, 9770, 9788, 9805, 9806, 9819, 9836, 9858, 9871, 9881, 9895, 9916, 9922, 9928, 9935, 9940, 9943, 9991, 9997, 10011, 10025, 10049, 10055, 10057, 10099, 10113, 10114, 10142, 10149, 10150, 10177, 10183, 10187, 10188, 10258, 10265, 10316, 10320, 10324, 10328, 10346, 10347, 10358, 10361, 10362, 10377– 10379, 10406–10410, 10437–10441	<code>\tex_holdinginserts:D</code> 395 <code>\tex_hrule:D</code> 331 <code>\tex_hsize:D</code> 356 <code>\tex_hskip:D</code> 321, 3858 <code>\tex_hss:D</code> 322, 5550, 5551 <code>\tex_ht:D</code> 460, 5467 <code>\tex_hyphen:D</code> 145, 791 <code>\tex_hyphenation:D</code> 446 <code>\tex_hyphenchar:D</code> 430 <code>\tex_hyphenpenalty:D</code> 348 <code>\tex_if:D</code> 184, 769, 866, 869, 1134, 7204, 7533, 7602, 7676 <code>\tex_ifcase:D</code> . 185, 1450, 3038, 8910, 8998 <code>\tex_ifcat:D</code> 186, 870, 2450, 2463 <code>\tex_ifdim:D</code> 189, 3971 <code>\tex_ifeof:D</code> 190, 5822, 5825 <code>\tex_iffalse:D</code> 195, 861, 4950, 4954, 5025, 5028 <code>\tex_ifhbox:D</code> 191, 5422, 5426 <code>\tex_ifhmode:D</code> 197, 873 <code>\tex_ifinner:D</code> 200, 875 <code>\tex_ifnum:D</code> 198, 872 <code>\tex_ifnum:D</code> 187, 1202, 1208, 1288, 1294, 3035, 3036, 3558, 3565, 3572, 7223, 7234, 7271, 7279, 7286, 7299, 7316, 7338, 7339, 7353, 7360, 7383, 7451, 7494, 7536, 7539, 7556, 7605, 7613, 7615, 7623, 7661, 7679, 7687, 7745, 7748, 7759, 7760, 7778, 7779, 7803– 7811, 7841–7849, 7898, 7907, 7935, 7944, 7972, 7980, 7984, 7993, 7994, 8000, 8046, 8083, 8103, 8128, 8143, 8147, 8149, 8218, 8222, 8315, 8325, 8361, 8365, 8395, 8399, 8406, 8409, 8411, 8425, 8456, 8461, 8471, 8475, 8479, 8480, 8602, 8612, 8637, 8647, 8668, 8707, 8711, 8725, 8730, 8731, 8748, 8752, 8756, 8758, 8782, 8797, 8807, 8840, 8853, 8879, 8894, 8939, 8967, 8982, 9008, 9009, 9013, 9020, 9034, 9035, 9056, 9088, 9089, 9093, 9099, 9109, 9113, 9139, 9152, 9176, 9186, 9187, 9193, 9200, 9201, 9230, 9231, 9260, 9414, 9447, 9449, 9450, 9460, 9474, 9485, 9501, 9502, 9523, 9533, 9550, 9551, 9580, 9581, 9587, 9616, 9619, 9654, 9658, 9662, 9668, 9678, 9682, 9747, 9748, 9798, 9801, 9815, 9831, 9837, 9862, 9875, 9885, 9909, 9912, 9923, 9931, 9987, 9993,
--	--

- 10005, 10015, 10034, 10044, 10050,
- 10080, 10084, 10101, 10118, 10123,
- 10126, 10153, 10156, 10160, 10161,
- 10306–10309, 10331, 10334, 10340,
- 10349, 10352, 10365, 10369, 10373,
- 10382, 10386, 10390, 10394, 10399,
- 10413, 10417, 10421, 10425, 10430
- `\tex_ifodd:D` 188, 649, 653, 778,
- 867, 868, 1264, 3037, 8786, 9938, 9941
- `\tex_iftrue:D` 196, 860
- `\tex_ifvbox:D` 192, 5423, 5429
- `\tex_ifvmode:D` 199, 874
- `\tex_ifvoid:D` 193, 5424, 5440
- `\tex_ifx:D` 194, 644,
- 698, 724, 797, 822, 871, 1778, 2435,
- 2441, 4274, 4807, 4983, 10254, 10261
- `\tex_ignorespaces:D` 242
- `\tex_immediate:D` 204, 735,
- 743, 806, 1234, 1237, 5634, 5750, 5789
- `\tex_indent:D` 338
- `\tex_input:D` 212, 785, 825, 7091
- `\tex_inputlineno:D` . 214, 1257, 1748, 5895
- `\tex_insert:D` 394
- `\tex_insertpenalties:D` 397
- `\tex_interlinepenalty:D` 376
- `\tex_italiccor:D` 144, 790
- `\tex_jobname:D` 451, 4244, 7015
- `\tex_kern:D` 329
- `\tex_language:D` 246
- `\tex_lastbox:D` 403, 5454
- `\tex_lastkern:D` 336
- `\tex_lastpenalty:D` 442
- `\tex_lastskip:D` 337
- `\tex_lccode:D` 465, 1142, 1757, 1758, 2305,
- 2318, 2579, 2583, 2586, 4410, 4411
- `\tex_leaders:D` 333
- `\tex_left:D` 301
- `\tex_lefthyphenmin:D` 357
- `\tex_leftskip:D` 359
- `\tex_leqno:D` 276
- `\tex_let:D` 122, 126,
- 132, 140, 146, 785–796, 812–814, 859
- `\tex_limits:D` 293
- `\tex_linepenalty:D` 349
- `\tex_lineskip:D` 343
- `\tex_lineskiplimit:D` 344
- `\tex_long:D` 165, 896
- `\tex_looseness:D` 361
- `\tex_lower:D` 398, 5462
- `\tex_lowercase:D` .. 437, 1144, 1766, 4333
- `\tex_mag:D` 245
- `\tex_mark:D` 247
- `\tex_mathaccent:D` 258
- `\tex_mathbin:D` 288
- `\tex_mathchar:D` 259
- `\tex_mathchardef:D` 156, 1232, 3681, 5412, 10541
- `\tex_mathchoice:D` 256
- `\tex_mathclose:D` 289
- `\tex_mathcode:D` 467, 2563, 2567, 2571, 2575
- `\tex_mathinner:D` 290
- `\tex_mathop:D` 291
- `\tex_mathopen:D` 295
- `\tex_mathord:D` 296
- `\tex_mathpunct:D` 297
- `\tex_mathrel:D` 298
- `\tex_mathsurround:D` 309
- `\tex_maxdeadcycles:D` 379
- `\tex_maxdepth:D` 380
- `\tex_meaning:D` 439, 882, 886, 1771
- `\tex_medmuskip:D` 310
- `\tex_message:D` 216, 847
- `\tex_mkern:D` 263
- `\tex_month:D` 449
- `\tex_moveleft:D` 399, 5459
- `\tex_moveright:D` 400, 5460
- `\tex_mskip:D` 260
- `\tex_multiply:D` 161, 8182, 8391, 8599, 8613, 9645
- `\tex_muskip:D` 457
- `\tex_muskipdef:D` 155, 4033
- `\tex_newlinechar:D` 211, 5812
- `\tex_noalign:D` 179
- `\tex_noboundary:D` 314
- `\tex_noexpand:D` 172, 666, 879, 1577
- `\tex_noindent:D` 340
- `\tex_nolimits:D` 294
- `\tex_nonscript:D` 274
- `\tex_nonstopmode:D` 237
- `\tex_nulldelimiterspace:D` 307
- `\tex_nullfont:D` 425
- `\tex_number:D` 434, 2094, 2097,
- 2116, 2119, 2121, 2122, 3032, 3304,
- 3972, 7664, 7765, 7784, 8111, 8236,
- 8379, 8848, 8902, 8947, 8990, 9147,
- 9268, 9422, 9636, 9780, 9823, 10198
- `\tex_omit:D` 180
- `\tex_openin:D` 206, 5647
- `\tex_openout:D` 207, 5634

<code>\tex_or:D</code>	203, 862, 8911, 8913, 8915, 8917, 8999, 9001, 9003, 9005	<code>\tex_scriptscriptfont:D</code>	428
<code>\tex_outer:D</code>	166	<code>\tex_scriptscriptstyle:D</code>	273
<code>\tex_output:D</code>	381	<code>\tex_scriptspace:D</code>	313
<code>\tex_outputpenalty:D</code>	391	<code>\tex_scriptstyle:D</code>	272
<code>\tex_over:D</code>	268	<code>\tex_scrollmode:D</code>	238
<code>\tex_overfullrule:D</code>	419	<code>\tex_setbox:D</code>	
<code>\tex_overline:D</code>	299	409, 5446, 5448, 5455, 5498, 5502, 5507, 5513, 5521, 5528, 5533, 5539	
<code>\tex_overwithdelims:D</code>	269	<code>\tex_setlanguage:D</code>	167
<code>\tex_pagedepth:D</code>	383	<code>\tex_sfcode:D</code>	464, 2599, 2603, 2607
<code>\tex_pagefilllstretch:D</code>	387	<code>\tex_shipout:D</code>	374, 849
<code>\tex_pagefillstretch:D</code>	386	<code>\tex_show:D</code>	217, 888
<code>\tex_pagefilstretch:D</code>	385	<code>\tex_showbox:D</code>	219, 5489
<code>\tex_pagegoal:D</code>	389	<code>\tex_showboxbreadth:D</code>	233
<code>\tex_pageshrink:D</code>	388	<code>\tex_showboxdepth:D</code>	234
<code>\tex_pagestretch:D</code>	384	<code>\tex_showlists:D</code>	220
<code>\tex_pagetotal:D</code>	390	<code>\tex_showthe:D</code>	
<code>\tex_par:D</code>	339, 796	218, 1404, 2526, 2575, 2586, 2596, 2607	
<code>\tex_parfillskip:D</code>	370	<code>\tex_skewchar:D</code>	431
<code>\tex_parindent:D</code>	363	<code>\tex_skip:D</code>	455
<code>\tex_parshape:D</code>	355	<code>\tex_skipdef:D</code>	154, 2740, 3791, 3876
<code>\tex_parskip:D</code>	362	<code>\tex_space:D</code>	143
<code>\tex_patterns:D</code>	445	<code>\tex_spacefactor:D</code>	373
<code>\tex_pausing:D</code>	232	<code>\tex_spaceskip:D</code>	368
<code>\tex_penalty:D</code>	440	<code>\tex_span:D</code>	181
<code>\tex_postdisplaypenalty:D</code>	287	<code>\tex_special:D</code>	443
<code>\tex_predisplaypenalty:D</code>	286	<code>\tex_splitbotmark:D</code>	252
<code>\tex_predisplaysize:D</code>	285	<code>\tex_splitfirstmark:D</code>	251
<code>\tex_pretolerance:D</code>	366	<code>\tex_splitmaxdepth:D</code>	421
<code>\tex_prevdepth:D</code>	413	<code>\tex_splittopskip:D</code>	422
<code>\tex_prevgraf:D</code>	372	<code>\tex_string:D</code>	436, 883
<code>\tex_radical:D</code>	261	<code>\tex_tabskip:D</code>	182
<code>\tex_raise:D</code>	401, 5461	<code>\tex_textfont:D</code>	426
<code>\tex_read:D</code>	208, 5829	<code>\tex_textstyle:D</code>	271
<code>\tex_relax:D</code>		<code>\tex_the:D</code>	
243, 645, 649, 653, 656–665, 669– 678, 683, 684, 687, 688, 692, 720, 721, 723, 730, 731, 738, 739, 778, 823, 825, 891, 1170, 1187, 1232, 1450, 1484, 1628, 1640, 1641, 2305, 2306, 2318, 2319, 3034, 3974, 4076		.. 244, 656–665, 1257, 1484, 1634, 1642, 1748, 3141, 3868, 3873, 3956, 3969, 4047, 4610, 7014, 7026, 10566	
<code>\tex_relpenalty:D</code>	304	<code>\tex_thickmuskip:D</code>	312
<code>\tex_right:D</code>	302	<code>\tex_thinmuskip:D</code>	311
<code>\tex_righthyphenmin:D</code>	358	<code>\tex_time:D</code>	447
<code>\tex_rightskip:D</code>	360	<code>\tex_toks:D</code>	456
<code>\tex_romannumeral:D</code>	435, 981, 983, 1134, 1140, 1148, 1602, 1613, 1621, 1710, 1713, 1716, 1802, 1809, 1814, 1892, 3039, 3515, 3984	<code>\tex_toksdef:D</code>	157, 2762, 4601, 4737
<code>\tex_scriptfont:D</code>	427	<code>\tex_tolerance:D</code>	367
		<code>\tex_topmark:D</code>	248
		<code>\tex_topskip:D</code>	378
		<code>\tex_tracingcommands:D</code>	223
		<code>\tex_tracinglostchars:D</code>	224
		<code>\tex_tracingmacros:D</code>	225
		<code>\tex_tracingonline:D</code>	226

<code>\tex_tracingoutput:D</code>	227	<code>\thepage</code>	6317
<code>\tex_tracingpages:D</code>	228	<code>\thickmuskip</code>	312
<code>\tex_tracingparagraphs:D</code>	229	<code>\thinmuskip</code>	311
<code>\tex_tracingrestores:D</code>	230	<code>\time</code>	447
<code>\tex_tracingstats:D</code>	231	<code>\tl_clear:c</code>	77, 4132 , 4763, 6639
<code>\tex_uccode:D</code>	466, 2589, 2593, 2596	<code>\tl_clear:N</code>	77, 4132 , 4132 , 4133, 4140, 4146, 4517, 4762, 5081, 5101, 6438, 6449, 7047, 7962, 8250, 8268, 8499, 8523, 8542, 8572, 8586
<code>\tex_uchyph:D</code>	364	<code>\tl_clear_new:c</code>	77, 4136 , 4767
<code>\tex_underline:D</code>	300, 786	<code>\tl_clear_new:N</code>	77, 4136 , 4137, 4146, 4147, 4766, 5085
<code>\tex_unhbox:D</code>	405, 5554	<code>\tl_const:Nn</code>	76, 2425, 2907, 4071 , 4071, 4245, 4246, 4416, 5567, 5842–5844, 5848, 5852, 5855, 5858, 5861, 5866, 5870, 5880, 5883–5885, 6358, 6555–6558
<code>\tex_unhcopy:D</code>	406, 5552	<code>\tl_elt_count:N</code>	82, 4393 , 4399
<code>\tex_unkern:D</code>	330	<code>\tl_elt_count:n</code>	82, 4393 , 4393 , 4398
<code>\tex_unpenalty:D</code>	441	<code>\tl_elt_count:o</code>	82, 4393
<code>\tex_unskip:D</code>	328	<code>\tl_elt_count:V</code>	82, 4393
<code>\tex_unvbox:D</code>	407, 5525	<code>\tl_elt_count_aux:n</code>	4395, 4401, 4404
<code>\tex_unvcopy:D</code>	408, 5523	<code>\tl_gclear:c</code>	77, 4132 , 4765
<code>\tex_uppercase:D</code>	438, 4334	<code>\tl_gclear:N</code>	77, 4132 , 4134, 4135, 4152, 4157, 4764, 5083
<code>\tex_vadjust:D</code>	341	<code>\tl_gclear_new:c</code>	77, 4148 , 4769
<code>\tex_valign:D</code>	176	<code>\tl_gclear_new:N</code>	77, 4148 , 4149, 4157, 4158, 4768, 5087
<code>\tex_vbadness:D</code>	416	<code>\tl_gput_left:cn</code>	78, 4198
<code>\tex_vbox:D</code>	411, 5496, 5498, 5507, 5513, 5518, 5519	<code>\tl_gput_left:co</code>	78
<code>\tex_vcenter:D</code>	262	<code>\tl_gput_left:cV</code>	78, 4198
<code>\tex_vfil:D</code>	323	<code>\tl_gput_left:cv</code>	4198
<code>\tex_vfill:D</code>	325	<code>\tl_gput_left:cx</code>	4198
<code>\tex_vfilneg:D</code>	324	<code>\tl_gput_left:Nn</code>	78, 4198 , 4213, 4232, 4793, 5171
<code>\tex_vfuzz:D</code>	418	<code>\tl_gput_left:No</code>	78, 4198 , 4222
<code>\tex_voffset:D</code>	393	<code>\tl_gput_left:Nv</code>	78, 4198 , 4216, 4233
<code>\tex_vrule:D</code>	332	<code>\tl_gput_left:Nv</code>	4198 , 4219, 4234
<code>\tex_vsize:D</code>	375	<code>\tl_gput_left:Nx</code>	78, 4198 , 4225, 4235
<code>\tex_vskip:D</code>	326, 3863	<code>\tl_gput_right:cn</code>	78, 4159
<code>\tex_vsplit:D</code>	404, 5521	<code>\tl_gput_right:co</code>	78, 4159
<code>\tex_vss:D</code>	327	<code>\tl_gput_right:cV</code>	78, 4159
<code>\tex_vtop:D</code>	412, 5497, 5502	<code>\tl_gput_right:cv</code>	4159
<code>\tex_wd:D</code>	459, 5469	<code>\tl_gput_right:cx</code>	4159
<code>\tex_widowpenalty:D</code>	346	<code>\tl_gput_right:Nn</code>	78, 4159 , 4174, 4193, 4795, 5179
<code>\tex_write:D</code>	209, 735, 743, 806, 881, 5814	<code>\tl_gput_right:No</code>	78, 4159 , 4183, 4196
<code>\tex_xdef:D</code>	150, 807, 919	<code>\tl_gput_right:Nv</code>	78, 4159 , 4177, 4194
<code>\tex_xleaders:D</code>	335	<code>\tl_gput_right:Nv</code>	4159 , 4180, 4195
<code>\tex_xspaceskip:D</code>	369	<code>\tl_gput_right:Nx</code>	78, 4159 , 4186, 4197
<code>\tex_year:D</code>	450		
<code>\textasteriskcentered</code>	3279, 3285		
<code>\textbardbl</code>	3284		
<code>\textdagger</code>	3280, 3286		
<code>\textdaggerdbl</code>	3281, 3287		
<code>\textfont</code>	426		
<code>\textparagraph</code>	3283		
<code>\textsection</code>	3282		
<code>\textstyle</code>	271		
<code>\TeXETstate</code>	526		
<code>\the</code>	100–109, 244		

- \tl_gremove_all_in:cn 84, [4545](#)
- \tl_gremove_all_in:Nn 84, [4545](#), [4548](#), [4552](#)
- \tl_gremove_in:cn 84, [4541](#)
- \tl_gremove_in:Nn .. 84, [4541](#), [4542](#), [4544](#)
- \tl_greplace_all_in:cnn 84, [4510](#)
- \tl_greplace_all_in:Nnn
..... 84, [4513](#), [4540](#), [4549](#)
- \tl_greplace_in:cnn 84, [4482](#)
- \tl_greplace_in:Nnn
..... 84, [4482](#), [4505](#), [4509](#), [4542](#)
- .tl_gset:c [139](#)
- \tl_gset:cn 77, [4087](#)
- \tl_gset:cV [4087](#)
- \tl_gset:cv [4087](#)
- \tl_gset:cx 77, [4087](#), [8892](#), [8980](#), [9258](#), [9813](#)
- .tl_gset:N [139](#)
- \tl_gset:Nc 77, [4236](#), [4236](#)
- \tl_gset:Nf [4087](#)
- \tl_gset:Nn 77, [4061](#), [4073](#), [4087](#),
[4093](#), [4108–4114](#), [4406](#), [4899](#), [5171](#),
[5179](#), [5258](#), [7090](#), [7435](#), [7891](#), [7927](#),
[8008](#), [8036](#), [8065](#), [8169](#), [8190](#), [8302](#),
[8826](#), [8925](#), [9125](#), [9399](#), [9735](#), [10063](#)
- \tl_gset:No 77, [4087](#)
- \tl_gset:NV 77, [4087](#)
- \tl_gset:Nv 77, [4087](#)
- \tl_gset:Nx 77, [4067](#),
[4087](#), [4096](#), [4115](#), [4449](#), [4781](#), [4940](#),
[5018](#), [5211](#), [5316](#), [7015](#), [7472](#), [9692](#)
- \tl_gset_eq:cc . 79, [4116](#), [4777](#), [5096](#), [7525](#)
- \tl_gset_eq:cN . 79, [4116](#), [4776](#), [5094](#), [7523](#)
- \tl_gset_eq:Nc . 79, [4116](#), [4775](#), [5095](#), [7524](#)
- \tl_gset_eq:NN 79, [4116](#),
[4121](#), [4128](#), [4131](#), [4134](#), [4506](#), [4514](#),
[4774](#), [5093](#), [7019](#), [7408](#), [7427](#), [7522](#)
- \tl_gset_rescan:Nnn . 80, [4405](#), [4406](#), [4408](#)
- \tl_gset_rescan:Nno 80, [4405](#)
- \tl_gset_rescan:Nnx 80, [4434](#), [4444](#)
- .tl_gset_x:c [139](#)
- .tl_gset_x:N [139](#)
- \tl_head:n 85, [4553](#), [4553](#), [4554](#), [4563](#), [4564](#)
- \tl_head:V 85, [4553](#)
- \tl_head:v 85, [4553](#)
- \tl_head:w 85, [4553](#), [4553](#), [4559](#), [4560](#),
[4568](#), [4573](#), [4577](#), [4591](#), [7224](#), [7235](#)
- \tl_head_i:n 85, [4553](#), [4554](#)
- \tl_head_i:w 85, [4553](#), [4560](#)
- \tl_head_iii:f 85, [4553](#)
- \tl_head_iii:n 85, [4553](#), [4557](#), [4558](#)
- \tl_head_iii:w 85, [4553](#), [4557](#), [4562](#)
- \tl_if_blank:n [4296](#)
- \tl_if_blank:nF .. 2355, [3301](#), [4305](#), [4309](#)
- \tl_if_blank:nT [4304](#), [4308](#)
- \tl_if_blank:nTF 81, [4296](#), [4303](#), [4307](#), [4314](#)
- \tl_if_blank:oTF ... 81, [4296](#), [4325](#), [6455](#)
- \tl_if_blank:VTF 81, [4296](#)
- \tl_if_blank_p:n ... 81, [4296](#), [4302](#), [4306](#)
- \tl_if_blank_p:o [4296](#)
- \tl_if_blank_p:oTF 81
- \tl_if_blank_p:V [4296](#)
- \tl_if_blank_p:VTF 81
- \tl_if_blank_p_aux:w ... [4296](#), [4297](#), [4299](#)
- \tl_if_empty:c [4973](#), [5098](#)
- \tl_if_empty:cTF 80, [4255](#)
- \tl_if_empty:N [4255](#), [4971](#), [5097](#)
- \tl_if_empty:n [4284](#)
- \tl_if_empty:NF 2345, [2371](#), [4262](#), [5206](#), [7081](#)
- \tl_if_empty:nF 2394,
[2948](#), [4291](#), [4295](#), [5163](#), [5189](#), [5340](#)
- \tl_if_empty:NT [4261](#)
- \tl_if_empty:nT [4290](#), [4294](#), [5365](#)
- \tl_if_empty:NTF
[80](#), [4255](#), [4260](#), [4322](#), [6493](#), [6515](#), [7074](#)
- \tl_if_empty:nTF
..... 80, [2719](#), [2726](#), [2737](#), [2748](#),
[2759](#), [2770](#), [2777](#), [2784](#), [2792](#), [3342](#),
[4284](#), [4289](#), [4293](#), [4311](#), [4331](#), [5960](#)
- \tl_if_empty:oTF 80, [4284](#)
- \tl_if_empty:VTF 80, [4284](#), [4723](#)
- \tl_if_empty_p:c 80, [4255](#)
- \tl_if_empty_p:N 80, [4255](#), [4259](#)
- \tl_if_empty_p:n 80, [2689](#), [4284](#), [4288](#), [4292](#)
- \tl_if_empty_p:o 80, [4284](#)
- \tl_if_empty_p:V 80, [4284](#)
- \tl_if_eq:cc [5108](#)
- \tl_if_eq:ccTF 80, [4263](#), [6718](#)
- \tl_if_eq:cN [5106](#)
- \tl_if_eq:cNTF 80, [4263](#)
- \tl_if_eq:Nc [5107](#)
- \tl_if_eq:NcTF 80, [4263](#)
- \tl_if_eq:NN [4263](#), [5105](#)
- \tl_if_eq:nn [4270](#)
- \tl_if_eq:NNF [4269](#), [5949](#)
- \tl_if_eq:NNT [4268](#), [5030](#)
- \tl_if_eq:NNTF
[80](#), [2221](#), [4263](#), [4267](#), [6066](#), [6469](#), [6520](#)
- \tl_if_eq:nnTF 81, [4270](#)
- \tl_if_eq_p:cc 80, [4263](#)
- \tl_if_eq_p:cN 80, [4263](#)
- \tl_if_eq_p:Nc 80, [4263](#)

\tl_if_eq_p:N	80, 4263 , 4263	\tl_map_variable:Nn	82, 4371 , 4374 , 4375
\tl_if_head_eq_catcode:N	4589	\tl_map_variable:Nn	82, 4371 , 4371 , 4374
\tl_if_head_eq_catcode:NNTF	86, 4567	\tl_map_variable_aux:Nn	4376
\tl_if_head_eq_catcode_p:N	86, 4567	\tl_map_variable_aux:Nnn	4372, 4376, 4379
\tl_if_head_eq_charcode:fNTF	85, 3447 , 3460 , 4567	\tl_new:c	76, 4056 , 4070, 4761, 6638, 6643, 8891, 8979, 9257, 9275, 9278, 9281, 9284, 9287, 9290, 9293, 9296, 9299, 9302, 9305, 9308, 9311, 9314, 9317, 9320, 9323, 9326, 9329, 9332, 9335, 9338, 9341, 9344, 9347, 9350, 9353, 9356, 9359, 9362, 9365, 9368, 9371, 9374, 9377, 9380, 9383, 9386, 9389, 9392, 9695, 9698, 9701, 9704, 9707, 9710, 9713, 9716, 9719, 9722, 9725, 9728, 9812
\tl_if_head_eq_charcode:n	4571	\tl_new:cn	76, 4056 , 4070, 6256
\tl_if_head_eq_charcode:nNF	4588	\tl_new:N	76, 2873, 2874, 2876, 2906, 4056 , 4069, 4072, 4142, 4154, 4243, 4247–4254, 4282, 4283, 4482, 4758–4760, 5041, 5079, 5886–5888, 5893, 6356, 6357, 6359–6361, 6560–6565, 6567, 7010, 7030, 7142, 7144, 7146, 7148, 7150, 7152, 7182, 7189, 7192, 7195, 7407, 9691, 10593
\tl_if_head_eq_charcode:nNTF	85, 4567 , 4586	\tl_new:Nn	76, 4056 , 4056, 4063, 4069
\tl_if_head_eq_charcode_p:fN	85, 4567 , 4576 , 4584	\tl_new:Nx	76, 4056 , 4064
\tl_if_head_eq_charcode_p:n	85, 4567 , 4585	\tl_num_elt_count_aux:n	4404
\tl_if_head_eq_meaning:N	4567	\tl_put_left:cn	78, 4198
\tl_if_head_eq_meaning:NNTF	85, 4567 , 6484	\tl_put_left:co	78
\tl_if_head_eq_meaning_p:N	85, 4567	\tl_put_left:cV	78, 4198
\tl_if_in:cnTF	84, 4461	\tl_put_left:cv	4198
\tl_if_in:Nn	4461	\tl_put_left:cx	4198
\tl_if_in:nn	4470	\tl_put_left:Nn	78, 4198 , 4198 , 4228, 4785, 5167
\tl_if_in:NnF	4469	\tl_put_left:No	78, 4198 , 4210
\tl_if_in:nnF	4478, 4481	\tl_put_left:NV	78, 4198 , 4201 , 4229
\tl_if_in:NnT	4468	\tl_put_left:Nv	4198 , 4204 , 4230
\tl_if_in:nnT	4477, 4480	\tl_put_left:Nx	78, 4198 , 4207 , 4231
\tl_if_in:NnTF	84, 4461 , 4467	\tl_put_right:cn	78, 4159
\tl_if_in:nnTF	84, 4470 , 4476 , 4479 , 6733 , 6743	\tl_put_right:co	78
\tl_if_in:onTF	84, 4470	\tl_put_right:cV	78, 4159
\tl_if_in:VnTF	84, 4470	\tl_put_right:cv	4159
\tl_if_single:N	4321	\tl_put_right:cx	4159
\tl_if_single:n	4310	\tl_put_right:Nn	78, 4159 , 4159 , 4189, 4787, 5175, 6494, 6517, 6522
\tl_if_single:NNTF	81	\tl_put_right:No	78, 4159 , 4171, 4493, 4523, 4527, 6417, 6471, 6489, 6500, 6505
\tl_if_single:NNTF	81, 4310	\tl_put_right:NV	78, 4159 , 4162 , 4190
\tl_if_single_p:n	81, 4310	\tl_put_right:Nv	4159 , 4165 , 4191
\tl_if_single_p:NNTF	81		
\tl_map_break:	82, 4381 , 4381 , 5655 , 5662		
\tl_map_function:cn	81, 4340		
\tl_map_function:Nn	81, 4340 , 4340 , 4395		
\tl_map_function_aux:Nn	4340		
\tl_map_function_aux:Nn	4341, 4344 , 4347 , 4349 , 4356 , 4365		
\tl_map_inline:cn	82, 4352		
\tl_map_inline:Nn	82, 4352 , 4361 , 4370		
\tl_map_inline:nn	82, 2710 , 4352 , 4352		
\tl_map_inline_aux:n	4352		
\tl_map_variable:cn	82, 4371		

- \tl_put_right:Nx . . . 78, [4159](#), 4168, 4192
- \tl_remove_all_in:cn 84, [4545](#)
- \tl_remove_all_in:Nn 84, [4545](#), 4545, 4551
- \tl_remove_in:cn 84, [4541](#)
- \tl_remove_in:Nn . . . 84, [4541](#), 4541, 4543
- \tl_replace_all_in:cn 84
- \tl_replace_all_in:Nnn
84, [4510](#), 4510, 4539, 4546, 6398, 6401
- \tl_replace_in:cn 84, [4482](#)
- \tl_replace_in:Nnn
. 84, [4482](#), 4483, 4508, 4541
- \tl_rescan:nn 79, [4452](#), 4452
- \tl_rescan_aux:w . 4423, [4428](#), 4429, 4457
- \tl_reverse:N 82, [4390](#), 4390
- \tl_reverse:n 82, [4382](#), 4382, 4389
- \tl_reverse:o 82, [4382](#), 4391
- \tl_reverse:V 82, [4382](#)
- \tl_reverse_aux:nN [4382](#), 4383, 4385, 4387
- .tl_set:c 139
- \tl_set:cn 77,
[4087](#), 6644, 9276, 9279, 9282, 9285,
9288, 9291, 9294, 9297, 9300, 9303,
9306, 9309, 9312, 9315, 9318, 9321,
9324, 9327, 9330, 9333, 9336, 9339,
9342, 9345, 9348, 9351, 9354, 9357,
9360, 9363, 9366, 9369, 9372, 9375,
9378, 9381, 9384, 9387, 9390, 9393,
9696, 9699, 9702, 9705, 9708, 9711,
9714, 9717, 9720, 9723, 9726, 9729
- \tl_set:co 77, [4087](#)
- \tl_set:cV 77, [4087](#)
- \tl_set:cv [4087](#)
- \tl_set:cx 77, [4087](#), 6585
- .tl_set:N 139
- \tl_set:Nc 77, [4236](#), 4241, 4242
- \tl_set:Nf 77, [4087](#), 4391, 6414, 6422
- \tl_set:Nn 77, 2879–2881,
2893, 2894, 2896, [4087](#), 4087, 4099–
4106, 4272, 4273, 4377, 4405, 4432,
4442, 4859, 4894, 4897, 4909, 4924,
4955, 4979, 4982, 5026, 5035, 5133,
5167, 5175, 5252, 5258, 5312, 5321,
6045, 6046, 6314, 6439, 6468, 6514,
6618, 6651, 6654, 6750, 6767, 6770,
6813, 7143, 7145, 7147, 7149, 7151,
7432, 7887, 7923, 8005, 8033, 8062,
8166, 8187, 8299, 8823, 8922, 9122,
9396, 9732, 10060, 10119, 10131, 10578
- \tl_set:No
77, [4087](#), 4242, 4491, 5202, 5203, 6429
- \tl_set:Nv 77, [4087](#), 6607, 6816
- \tl_set:Nx 77, [4087](#),
4090, 4107, 4244, 4439, 4779, 4938,
5016, 5055, 5210, 5939, 5940, 5961,
5963, 6488, 6732, 6744, 6749, 6782,
6783, 7042, 7061, 7214, 7226, 7237,
7329, 7373, 7396, 7469, 7989, 7992,
8044, 8295, 8650, 8659, 8680, 8697,
8838, 8937, 9137, 9412, 9492, 9524,
9774, 9838, 9888, 9897, 10018, 10444
- \tl_set_eq:cc
. . . . 79, [4116](#), 4773, 5092, 6823, 7521
- \tl_set_eq:cN . . 79, [4116](#), 4772, 5090, 7519
- \tl_set_eq:Nc . . 79, [4116](#), 4771, 5091, 7520
- \tl_set_eq:NN 79,
2895, [4116](#), 4117, 4127, 4130, 4132,
4484, 4511, 4770, 5089, 7424, 7518
- \tl_set_rescan:Nnn . . 80, [4405](#), 4405, 4407
- \tl_set_rescan:Nno 80, [4405](#), 7215
- \tl_set_rescan:Nnx 80, [4434](#), 4434
- \tl_set_rescan_aux:NNnn
. 4405, 4406, [4409](#), 4418
- .tl_set_x:c 139
- .tl_set_x:N 139
- \tl_show:c 77, [4084](#), 5145, 7527
- \tl_show:N 77, [4084](#), 4084, 4085, 5144, 7526
- \tl_show:n 77, [4084](#), 4086, 5047
- \tl_tail:f 85, [4553](#)
- \tl_tail:n 85, [4553](#), 4555, 4556, 4565, 4566
- \tl_tail:V 85, [4553](#)
- \tl_tail:v 85, [4553](#)
- \tl_tail:w 85, [4553](#), 4555, 4561, 7229, 7240
- \tl_tmp:w 4462, 4465, 4471, 4474
- \tl_to_lowercase:n
. . . . 81, 2307, 2320, 2712, 2802,
[4333](#), 4333, 4414, 5878, 5933, 6396
- \tl_to_str:c 79, [4336](#)
- \tl_to_str:N 79, [4336](#), 4336, 4339
- \tl_to_str:n 79, 4285, 4297, [4335](#), 4335,
6588, 6745, 6749, 6782, 6809, 7511
- \tl_to_str_aux:w [4336](#), 4336, 4338
- \tl_to_uppercase:n 81, [4333](#), 4334
- \tl_use:c 76, [4075](#)
- \tl_use:N 76, [4075](#), 4075, 4083, 6802
- \token_get_arg_spec:N . . . 53, [2795](#), 2813
- \token_get_prefix_arg_replacement_aux:w
. [2795](#), 2804, 2809, 2815, 2821
- \token_get_prefix_spec:N . 53, [2795](#), 2807

\token_get_replacement_spec:N	\token_if_long_macro:N	2779		
.....	53, 2795, 2819	\token_if_long_macro:NTF	52, 2703		
\token_if_active_char:N	2669	\token_if_long_macro_aux:w	.. 2780, 2783		
\token_if_active_char:NTF	51, 2669	\token_if_long_macro_p:N	52, 2703	
\token_if_active_char_p:N	51, 2669, 2834	\token_if_long_macro_p_aux:w	2703		
\token_if_alignment_tab:N	2641	\token_if_macro:N	2685	
\token_if_alignment_tab:NTF	... 50, 2641	\token_if_macro:NTF	51, 2685, 2808, 2814, 2820	
\token_if_alignment_tab_p:N	... 50, 2641	\token_if_macro_p:N	. 51, 2685, 2828, 2835	\token_if_macro_p_aux:w	2685, 2686, 2688	
\token_if_chardef:N	2714	\token_if_math_shift:N	2637	
\token_if_chardef:NTF	52, 2703	\token_if_math_shift:NTF	50, 2637	
\token_if_chardef_aux:w	2715, 2718	\token_if_math_shift_p:N	50, 2637	
\token_if_chardef_p:N	... 52, 2703, 2846	\token_if_math_subscript:N	2653		
\token_if_chardef_p_aux:w	2703	\token_if_math_subscript:NTF	.. 50, 2653		
\token_if_cs:N	2692	\token_if_math_subscript_p:N	.. 50, 2653		
\token_if_cs:NTF	51, 2692	\token_if_math_superscript:N	2649	
\token_if_cs_p:N	51, 2692, 2827	\token_if_math_superscript:NTF	50, 2649		
\token_if_dim_register:N	2750	\token_if_math_superscript_p:N	50, 2649		
\token_if_dim_register:NTF	52, 2703	\token_if_mathchardef:N	2721	
\token_if_dim_register_aux:w	2754, 2758	\token_if_mathchardef:NTF	52, 2703		
\token_if_dim_register_p:N	52, 2703, 2854	\token_if_mathchardef_aux:w	. 2722, 2725			
\token_if_dim_register_p_aux:w	.. 2703	\token_if_mathchardef_p:N	52, 2703, 2848			
\token_if_eq_catcode:NN	2677	\token_if_mathchardef_p_aux:w	... 2703		
\token_if_eq_catcode:NNTF	51, 2677	\token_if_other_char:N	2665	
\token_if_eq_catcode_p:NN	51, 2677, 2693, 2926, 2927	\token_if_other_char:NTF	50, 2665
\token_if_eq_charcode:NN	2681	\token_if_eq_charcode:NNTF	51, 2681	
\token_if_eq_charcode:NNTF	51, 2681	\token_if_eq_charcode_p:NN	51, 2681	
\token_if_eq_charcode_p:NN	51, 2681	\token_if_eq_meaning:NN	2673	
\token_if_eq_meaning:NN	2673	\token_if_eq_meaning:NNT	2314	
\token_if_eq_meaning:NNT	2314	\token_if_eq_meaning:NNTF	51, 2327, 2673, 3019
\token_if_eq_meaning:NNTF	51, 2327, 2673, 3019	\token_if_eq_meaning_p:NN	51, 2673, 2928	
\token_if_eq_meaning_p:NN	51, 2673, 2928	\token_if_expandable:N	2695		
\token_if_expandable:N	2695	\token_if_expandable:NTF	51, 2695	
\token_if_expandable:NTF	51, 2695	\token_if_expandable_p:N	51, 2695	
\token_if_expandable_p:N	51, 2695	\token_if_group_begin:N	2629	
\token_if_group_begin:N	2629	\token_if_group_begin:NTF	49, 2629	
\token_if_group_begin:NTF	49, 2629	\token_if_group_begin_p:N	49, 2629	
\token_if_group_begin_p:N	49, 2629	\token_if_group_end:N	2633	
\token_if_group_end:N	2633	\token_if_group_end:NTF	49, 2633	
\token_if_group_end:NTF	49, 2633	\token_if_group_end_p:N	49, 2633	
\token_if_group_end_p:N	49, 2633	\token_if_int_register:N	2728	
\token_if_int_register:N	2728	\token_if_int_register:NTF	52, 2703	
\token_if_int_register:NTF	52, 2703	\token_if_int_register_aux:w	2732, 2736		
\token_if_int_register_aux:w	2732, 2736	\token_if_int_register_p:N	52, 2703, 2850			
\token_if_int_register_p:N	52, 2703, 2850	\token_if_int_register_p_aux:w	.. 2703			
\token_if_int_register_p_aux:w	.. 2703	\token_if_letter:N	2661		
\token_if_letter:N	2661	\token_if_letter:NTF	50, 2661	
\token_if_letter:NTF	50, 2661	\token_if_letter_p:N	50, 2661	
\token_if_letter_p:N	50, 2661	\token_if_long_macro:N	2779	
\token_if_long_macro:N	2779	\token_if_long_macro:NTF	52, 2703	
\token_if_long_macro:NTF	52, 2703	\token_if_long_macro_aux:w	... 2780, 2783		
\token_if_long_macro_aux:w	... 2780, 2783	\token_if_long_macro_p:N	52, 2703		
\token_if_long_macro_p:N	52, 2703	\token_if_long_macro_p_aux:w	2703	
\token_if_long_macro_p_aux:w	2703	\token_if_macro:N	2685	
\token_if_macro:N	2685	\token_if_macro:NTF	51, 2685, 2808, 2814, 2820
\token_if_macro:NTF	51, 2685, 2808, 2814, 2820	\token_if_macro_p:N	. 51, 2685, 2828, 2835	
\token_if_macro_p:N	. 51, 2685, 2828, 2835	\token_if_macro_p_aux:w	2685, 2686, 2688			
\token_if_macro_p_aux:w	2685, 2686, 2688	\token_if_math_shift:N	2637		
\token_if_math_shift:N	2637	\token_if_math_shift:NTF	50, 2637	
\token_if_math_shift:NTF	50, 2637	\token_if_math_shift_p:N	50, 2637	
\token_if_math_shift_p:N	50, 2637	\token_if_math_subscript:N	2653	
\token_if_math_subscript:N	2653	\token_if_math_subscript:NTF	.. 50, 2653		
\token_if_math_subscript:NTF	.. 50, 2653	\token_if_math_subscript_p:N	.. 50, 2653			
\token_if_math_subscript_p:N	.. 50, 2653	\token_if_math_superscript:N	2649		
\token_if_math_superscript:N	2649	\token_if_math_superscript:NTF	50, 2649		
\token_if_math_superscript:NTF	50, 2649	\token_if_math_superscript_p:N	50, 2649			
\token_if_math_superscript_p:N	50, 2649	\token_if_mathchardef:N	2721		
\token_if_mathchardef:N	2721	\token_if_mathchardef:NTF	52, 2703	
\token_if_mathchardef:NTF	52, 2703	\token_if_mathchardef_aux:w	. 2722, 2725		
\token_if_mathchardef_aux:w	. 2722, 2725	\token_if_mathchardef_p:N	52, 2703, 2848			
\token_if_mathchardef_p:N	52, 2703, 2848	\token_if_mathchardef_p_aux:w	... 2703			
\token_if_mathchardef_p_aux:w	... 2703	\token_if_other_char:N	2665		
\token_if_other_char:N	2665	\token_if_other_char:NTF	50, 2665	
\token_if_other_char:NTF	50, 2665	\token_if_other_char_p:N	50, 2665	
\token_if_other_char_p:N	50, 2665	\token_if_parameter:N	2645	
\token_if_parameter:N	2645	\token_if_parameter:NTF	50, 2645	
\token_if_parameter:NTF	50, 2645	\token_if_parameter_p:N	50, 2645	
\token_if_parameter_p:N	50, 2645	\token_if_primitive:N	2826	
\token_if_primitive:N	2826	\token_if_primitive:NTF	53, 2826	
\token_if_primitive:NTF	53, 2826	\token_if_primitive_p:N	53, 2826	
\token_if_primitive_p:N	53, 2826	\token_if_primitive_p_aux:N	2826, 2831, 2838, 2845
\token_if_primitive_p_aux:N	2826, 2831, 2838, 2845	\token_if_protected_long_macro:N	. 2786	
\token_if_protected_long_macro:N	. 2786	\token_if_protected_long_macro:NTF	52, 2703	
\token_if_protected_long_macro:NTF	52, 2703	\token_if_protected_long_macro_aux:w	2787, 2790
\token_if_protected_long_macro_aux:w	2787, 2790	\token_if_protected_long_macro_p:N	52, 2703
\token_if_protected_long_macro_p:N	52, 2703	\token_if_protected_long_macro_p_aux:w	2703
\token_if_protected_long_macro_p_aux:w	2703	\token_if_protected_macro:N	2772	
\token_if_protected_macro:N	2772	\token_if_protected_macro:NTF	. 52, 2703		
\token_if_protected_macro:NTF	. 52, 2703	\token_if_protected_macro_aux:w	2773, 2776	
\token_if_protected_macro_aux:w	2773, 2776	\token_if_protected_macro_p:N	. 52, 2703	
\token_if_protected_macro_p:N	. 52, 2703	\token_if_protected_macro_p_aux:w	2703			
\token_if_protected_macro_p_aux:w	2703					

\token_if_skip_register:N 2739
\token_if_skip_register:NTF ... 53, 2703
\token_if_skip_register_aux:w 2743, 2747
\token_if_skip_register_p:N
..... 53, 2703, 2852
\token_if_skip_register_p_aux:w .. 2703
\token_if_space:N 2657
\token_if_space:NTF 50, 2657
\token_if_space_p:N 50, 2657
\token_if_toks_register:N 2761
\token_if_toks_register:NTF ... 53, 2703
\token_if_toks_register_aux:w 2765, 2769
\token_if_toks_register_p:N
..... 53, 2703, 2856
\token_if_toks_register_p_aux:w .. 2703
\token_new:Nn ... 49, 2611, 2611, 2616,
2618–2620, 2622–2625, 2866–2868
\token_to_meaning:N 24,
881, 882, 1250, 1271, 2686, 2716,
2723, 2733, 2744, 2755, 2766, 2774,
2781, 2788, 2810, 2816, 2822, 4337
\token_to_str:c . 24, 881, 890, 1745, 2339
\token_to_str:N 5,
24, 881, 883, 890, 1134, 1137, 1249,
1255, 1256, 1270, 1280, 1407, 1483,
2324, 4077, 4808, 5046, 5052, 5102,
5122, 5147, 5287, 6966, 7419, 10550
\toks 456
\toks_clear:c 88, 4651, 5273
\toks_clear:N 88, 4651, 4651,
4658, 4660, 4663, 5149, 5272, 5289
\toks_gclear:c 88, 4651, 5275
\toks_gclear:N . 88, 4651, 4655, 4661, 5274
\toks_gput_left:cn 89, 4674
\toks_gput_left:co 89, 4674
\toks_gput_left:cV 89, 4674
\toks_gput_left:Nn .. 89, 4674, 4679, 4684
\toks_gput_left:Nv 89, 4674
\toks_gput_left:Nx 89, 4674
\toks_gput_right:cn 89, 4689
\toks_gput_right:co 89, 4689
\toks_gput_right:cV 89, 4689
\toks_gput_right:Nn
..... 89, 4689, 4693, 4713, 5335
\toks_gput_right:No 89, 4689
\toks_gput_right:Nv 89, 4689
\toks_gput_right:Nx 89, 4689
\toks_gset:cn 87, 4630, 6287
\toks_gset:co 87, 4630
\toks_gset:cV 87, 4630
\toks_gset:cx 87, 4630
\toks_gset:Nn
..... 87, 4630, 4630–4632, 5319, 5351
\toks_gset:No 87, 4630
\toks_gset:Nv 87, 4630
\toks_gset:Nx 87, 4630
\toks_gset_eq:cc 88, 4633, 5283
\toks_gset_eq:cN 88, 4633, 5282
\toks_gset_eq:Nc 88, 4633, 5281
\toks_gset_eq:NN
..... 88, 4633, 4639, 4647, 4650, 5280
\toks_if_empty:c 5367
\toks_if_empty:cTF 90, 4722
\toks_if_empty:N 4722, 5366
\toks_if_empty:Nf 4728, 5151, 5291
\toks_if_empty:NT 4727
\toks_if_empty:NTF 90, 4722, 4726
\toks_if_empty_p:c 4722
\toks_if_empty_p:cTF 90
\toks_if_empty_p:N 90, 4722, 4725
\toks_if_eq:cc 5371
\toks_if_eq:ccTF 90, 4729
\toks_if_eq:cN 5369
\toks_if_eq:cNTF 90, 4729
\toks_if_eq:Nc 5370
\toks_if_eq:NcTF 90, 4729
\toks_if_eq:NN 4729, 5368
\toks_if_eq:NNf 4736
\toks_if_eq:NNt 4735
\toks_if_eq:NNTF 90, 4729, 4734
\toks_if_eq_p:cc 4729
\toks_if_eq_p:ccTF 90
\toks_if_eq_p:cN 4729
\toks_if_eq_p:cNTF 90
\toks_if_eq_p:Nc 4729
\toks_if_eq_p:NcTF 90
\toks_if_eq_p:NN 4729, 4733
\toks_if_eq_p:NNTF 90
\toks_new:c 86, 4600, 5271
\toks_new:N
..... 86, 4600, 4604, 4609, 4739–4745, 5270
\toks_put_left:cn 89, 4674
\toks_put_left:co 89, 4674
\toks_put_left:cV 89, 4674
\toks_put_left:Nn 89, 4674, 4674, 4678, 4682
\toks_put_left:No 89, 4674
\toks_put_left:Nv 89, 4674
\toks_put_left:Nx 89, 4674
\toks_put_left_aux:w ... 4674, 4675, 4685

<code>\toks_put_right:cn</code>	89, 4689	<code>\tracingnesting</code>	486
<code>\toks_put_right:co</code>	89, 4689	<code>\tracingonline</code>	226
<code>\toks_put_right:cV</code>	89, 4689	<code>\tracingoutput</code>	227
<code>\toks_put_right:Nf</code>	89, 4714 , 4716	<code>\tracingpages</code>	228
<code>\toks_put_right:Nn</code>	89, 4689 , 4689, 4696, 4698, 4712, 4714, 5329	<code>\tracingparagraphs</code>	229
<code>\toks_put_right:No</code>	89, 4689 , 4706	<code>\tracingrestores</code>	230
<code>\toks_put_right:NV</code>	89, 4689 , 4700	<code>\tracingscantokens</code>	485
<code>\toks_put_right:Nx</code>	89, 4689 , 5152, 5154, 5292, 5294	<code>\tracingstats</code>	231
<code>\toks_set:cf</code>	87, 4612	U	
<code>\toks_set:cn</code>	87, 4612	<code>\U</code>	2710
<code>\toks_set:co</code>	87, 4612	<code>\uccode</code>	466
<code>\toks_set:cV</code>	87, 4612	<code>\uchyph</code>	364
<code>\toks_set:cv</code>	87, 4612	<code>\underline</code>	300
<code>\toks_set:cx</code>	87, 4612	<code>\unexpanded</code>	97, 479
<code>\toks_set:Nf</code>	87, 4612 , 4625	<code>\unhbox</code>	405
<code>\toks_set:Nn</code>	87, 4612 , 4613, 4614, 4617, 4629, 5349	<code>\unhcopy</code>	406
<code>\toks_set:No</code>	87, 4612 , 4624	<code>\unkern</code>	330
<code>\toks_set:NV</code>	87, 4420 , 4454 , 4612 , 4618	<code>\unless</code>	470
<code>\toks_set:Nv</code>	87, 4612 , 4621	<code>\unpenalty</code>	441
<code>\toks_set:Nx</code>	87, 4612 , 6257	<code>\unskip</code>	328
<code>\toks_set_eq:cc</code>	87, 4633 , 5279	<code>\unvbox</code>	407
<code>\toks_set_eq:cN</code>	87, 4633 , 5278	<code>\unvcopy</code>	408
<code>\toks_set_eq:Nc</code>	87, 4633 , 5277	<code>\uppercase</code>	438
<code>\toks_set_eq:NN</code>	87, 4633 , 4634, 4646, 4649, 5276	<code>\use:c</code>	13, 938 , 938, 1051, 1731, 1736, 1746, 2050, 2057, 2131, 2132, 3167, 3317, 3326, 3328, 3330, 3331, 3335, 3519, 3988, 6030, 6076, 6087, 6094, 6103, 6110, 6116, 6122, 6172, 6185, 6194, 6574, 6828, 7219, 7249, 7252, 7270, 7273, 7274, 7277, 7280, 7565, 7631, 7695, 7736, 8871, 8960, 9170, 9438, 9793, 10299, 10360, 10374, 10376, 10464
<code>\toks_show:c</code>	88, 4672 , 5285	<code>\use:n</code>	12, 946 , 946 , 1067, 1096, 1484, 2190, 2200, 2210, 2220, 3559, 3568, 4498, 4531, 4804
<code>\toks_show:N</code>	88, 4672 , 4672 , 4673 , 5158, 5284, 5300	<code>\use:nn</code>	12, 946 , 947
<code>\toks_use:c</code>	87, 4610	<code>\use:nnn</code>	12, 946 , 948
<code>\toks_use:N</code>	87, 4610 , 4610, 4611, 4676, 4690, 4702, 4708, 4718, 4730, 5307, 5383, 5895, 6263, 6269, 6291, 6295	<code>\use:nnnn</code>	12, 946 , 949
<code>\toks_use_clear:c</code>	88, 4662	<code>\use:x</code>	13, 939 , 939
<code>\toks_use_clear:N</code>	88, 4662 , 4662 , 4668 , 4670	<code>\use_0_parameter:</code>	1358
<code>\toks_use_gclear:c</code>	88, 4662	<code>\use_1_parameter:</code>	1358
<code>\toks_use_gclear:N</code>	88, 4662 , 4665 , 4671	<code>\use_2_parameter:</code>	1358
<code>\toksdef</code>	157	<code>\use_3_parameter:</code>	1358
<code>\tolerance</code>	367	<code>\use_4_parameter:</code>	1358
<code>\topmark</code>	248	<code>\use_5_parameter:</code>	1358
<code>\topmarks</code>	474	<code>\use_6_parameter:</code>	1358
<code>\topskip</code>	378	<code>\use_7_parameter:</code>	1358
<code>\tracingassigns</code>	484	<code>\use_8_parameter:</code>	1358
<code>\tracingcommands</code>	223		
<code>\tracinggroups</code>	491		
<code>\tracingifs</code>	487		
<code>\tracinglostchars</code>	224		
<code>\tracingmacros</code>	225		

`\use_9_parameter:` 1358
`\use_i:nn` 13, 950, 950,
 981, 1152, 1182, 1203, 1435, 3573,
 4686, 7318, 7362, 7385, 7964, 8290,
 8639, 8670, 9526, 9840, 9877, 10007
`\use_i:nnn` . 14, 952, 952, 1164, 2810, 9494
`\use_i:nnnn` 14, 952, 955
`\use_i_after_else:nw` 15, 966, 967
`\use_i_after_fi:nw` 15, 966, 966, 1471
`\use_i_after_or:nw` 15, 966, 968
`\use_i_after_orelse:nw` 15,
 966, 969, 1451, 1453, 1455, 1457,
 1459, 1461, 1463, 1465, 1467, 1469
`\use_i_delimit_by_q_nil:nw` . 14, 963, 963
`\use_i_delimit_by_q_recursion_stop:nw`
 14, 963, 965, 2442, 2471
`\use_i_delimit_by_q_stop:nw` 14, 963, 964
`\use_i_ii:nnn` 14, 952, 959, 1632
`\use_ii:nn` 13, 950, 951,
 983, 1154, 1184, 1205, 1437, 3575, 6458
`\use_ii:nnn` . 14, 952, 953, 1167, 2816, 6501
`\use_ii:nnnn` 14, 952, 956
`\use_iii:nnn` 14, 952, 954, 2822
`\use_iii:nnnn` 14, 952, 957
`\use_iv:nnnn` 14, 952, 958
`\use_none:n` 13,
 970, 970, 1067, 1096, 1137, 1139,
 1473, 1795, 2444, 2473, 3452, 3456,
 3461, 3561, 3566, 4756, 4820, 4923,
 4952, 5798, 5802, 6425, 6455, 7332,
 7376, 7399, 7458, 7501, 7914, 7951,
 8024, 8053, 8110, 8235, 8378, 8683,
 8700, 8847, 8901, 8946, 8989, 9146,
 9267, 9421, 9635, 9779, 9822, 10197
`\use_none:nn`
 13, 970, 971, 1754, 3157, 5031, 10250
`\use_none:nnn` 13, 970, 972, 6488
`\use_none:nnnn` 13, 970, 973, 7290
`\use_none:nnnnn` 13, 970, 974
`\use_none:nnnnnn` 13, 970, 975
`\use_none:nnnnnnn` 13, 970, 976
`\use_none:nnnnnnnn` 13, 970, 977
`\use_none:nnnnnnnnn` ... 13, 970, 978, 1441
`\use_none_delimit_by_q_nil:w` 14, 960, 960
`\use_none_delimit_by_q_recursion_stop:w`
 14, 960, 962, 1049,
 1117, 1728, 2416, 2436, 2458, 4381
`\use_none_delimit_by_q_stop:w`
 14, 960, 961, 2236, 2240, 5403
`\usepackage` 834, 6302

V

`\vadjust` 341, 629
`\valign` 176
`\value` 6320
`.value_forbidden:` 139
`.value_required:` 139
`\vbadness` 416
`\vbox` 411
`\vbox:n` 117, 5496, 5496
`\vbox_gset:cn` 117, 5498
`\vbox_gset:Nn` 117, 5498, 5500, 5501
`\vbox_gset_inline_begin:N` 118, 5512, 5515
`\vbox_gset_inline_end:` . 118, 5512, 5517
`\vbox_gset_to_ht:ccn` 118, 5506
`\vbox_gset_to_ht:cnn` 118, 5506
`\vbox_gset_to_ht:Nnn` 118, 5506, 5510, 5511
`\vbox_gset_top:cn` 117, 5502
`\vbox_gset_top:Nn` . 117, 5502, 5504, 5505
`\vbox_set:cn` 117, 5498
`\vbox_set:Nn` 117, 5498, 5498-5500
`\vbox_set_inline_begin:N`
 118, 5512, 5512, 5516
`\vbox_set_inline_end:` .. 118, 5512, 5514
`\vbox_set_split_to_ht:Nnn` 118, 5520, 5520
`\vbox_set_to_ht:cnn` 118, 5506
`\vbox_set_to_ht:Nnn`
 118, 5506, 5506, 5509, 5510
`\vbox_set_top:cn` 117, 5502
`\vbox_set_top:Nn` .. 117, 5502, 5502-5504
`\vbox_to_ht:nn` 118, 5518, 5518
`\vbox_to_zero:n` 118, 5518, 5519
`\vbox_top:n` 117, 5496, 5497
`\vbox_unpack:c` 118, 5523
`\vbox_unpack:N` 118, 5523, 5523, 5524
`\vbox_unpack_clear:c` 118, 5523
`\vbox_unpack_clear:N` 118, 5523, 5525, 5526
`\vcenter` 262
`\vfil` 323
`\vfill` 325
`\vfilneg` 324
`\vfuzz` 418
`\voffset` 393
`\voidb@x` 5491
`\vrule` 332
`\vsize` 375
`\vskip` 326
`\vsplit` 404
`\vss` 327
`\vtop` 412

W

	W		<code>\xref_define_label_aux:nn</code>	6279 , 6282 , 6284
<code>\WARNING</code>		6286	<code>\xref_get_value:nn</code>	
<code>\wd</code>		459		131 , 6265 , 6265 , 6315 , 6318 , 6321
<code>\widowpenalties</code>		519	<code>\xref_new:nn</code>	131 , 6249 , 6249 , 6313
<code>\widowpenalty</code>		346	<code>\xref_new_aux:nnn</code>	6249 , 6249 – 6251
<code>\write</code>		209	<code>\xref_set_label:n</code>	131 , 6290 , 6290 , 6323

X

	X			<code>\xref_tmp:w</code>	6291, 6294
<code>\X</code>	2706, 2710			<code>\xref_write</code>	380, 6292, 6304
<code>\xdef</code>	150			<code>\xspaceskip</code>	369
<code>\xetex_if_engine:</code>	1414, 1421				
<code>\xetex_if_engine:TF</code>	4, 25, 1411		Y		
<code>\xetex_version:D</code>	635, 1411		<code>\Y</code>	2707, 2710	
<code>\XeTeXversion</code>	635		<code>\year</code>	450	
<code>\xleaders</code>	335				
<code>\xref_deferred_new:nn</code>			Z		
	131, 6249, 6250, 6317, 6320		<code>\Z</code>	1757, 1765, 2708, 2710	
<code>\xref_define_label:nn</code> 380, 6279, 6279, 6293			<code>\z@</code>	3889	

Y

<code>\Y</code>	2707, 2710
<code>\year</code>	450

$$\mathbf{Z}$$

<code>\Z</code>	1757, 1765, 2708, 2710
<code>\z@</code>	3889